Naval Research Laboratory
Washington, DC 20375-5320

NRL/MR/5653--99-8419

An Introduction to ARRSTATS — A Computer Program for Simulating the Effects of Errors in Time- and Phase-Steered Planar Array Antennas

C. STAN WEST

Photonics Technology Branch
Optical Sciences Division

November 22, 1999

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave Blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	November 22, 1999	Final

4. TITLE AND SUBTITLE

An Introduction to ARRSTATS — A Computer Program for Simulating the Effects of Errors in Time- and Phase-Steered Planar Array Antennas

5. FUNDING NUMBERS

ONR — 63217N

6. AUTHOR(S)

C. Stan West

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Research Laboratory
Washington, DC 20375-5320

8. PERFORMING ORGANIZATION REPORT NUMBER

NRL/MR/5653--99-8419

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Bobby Junker
Office of Naval Research

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

A computer program is described that simulates time- and phase-steered array antennas subject to deterministic and random errors. The modeled array comprises a planar, rectangular grid of phase-steered elements grouped hierarchically into time-steered subarrays and time-steered subapertures. Random phase, time, and amplitude errors may be assigned, and the phases and times may be quantized. The simulation frequency may differ from the design frequency. The program assesses the antenna's performance by computing and analyzing the far-field radiation pattern or an ensemble of statistically identical patterns. It determines the location and power density of the main beam peak, the pointing error, the major and minor beam widths, the directivity, the ratio of main beam and sidelobe powers, and other measures. The program can tabulate measures as functions of a user-specified independent variable such as the operating frequency, a steering angle, an error parameter, a quantization specification, or a parameter of the array geometry. Results may be displayed as textual output of the performance measures, plots of radiation patterns, and plots of performance measures versus the independent variable.

14. SUBJECT TERMS

15. NUMBER OF PAGES

54

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std 239-18
298-102

CONTENTS

AN INTRODUCTION TO ARRSTATS — A COMPUTER PROGRAM FOR SIMULATING THE EFFECTS OF ERRORS IN TIME- AND PHASE-STEERED PLANAR ARRAY ANTENNAS

1. INTRODUCTION

Phased array antennas are remarkable for their suitability to many applications, which is partly because they steer quickly, allow adaptive processing, conform to special shapes, and produce a variety of radiation patterns. As low-sidelobe radiation patterns were being developed, it became apparent that errors in the phase or amplitude of the element excitations would limit the lowest achievable sidelobe level. Such errors might be due to manufacturing tolerances on the physical structure, inaccurate phase shifters, or non-uniform feeds, and can affect the average and peak sidelobe levels, beam width, pointing accuracy, and directivity, for example. Although one cannot predict the performance of a given antenna without measurements of the phase and amplitude accuracy of each element, one can draw conclusions about the behavior of an ensemble of antennas that are statistically identical but have different realizations of phase and amplitude errors. The present work applies this approach to a generalized array antenna that blends phase- and time-steering to achieve greater bandwidth at a reasonable cost.[1–3] The purpose of this report is to introduce the reader to a computer program for simulating time- and phase-steered planar array antennas subject to deterministic and random excitation errors. It is intended as an overview to the features and capabilities of the program and a guide to understanding the program's input, processing, and output.

The theoretical study of errors in array antennas has produced a large body of literature. Introductions to these works and key results and derivations may be found in several books.[4–6] The literature on simulations is more sparse, but two programs have been described recently against which this work may be contrasted. First, Chrisman[7] describes a program that simulates phase-steered planar arrays. It computes the directivity and cuts of the design and expected radiation patterns from the error statistics using theoretical formulas. One pattern cut intersects the main beam and boresight, and the other is normal to it through the main beam; from them the beam width is obtained in those directions. Second, Wright[8] briefly discusses the features of a simulation of phase-steered arrays. Its parameters include phase and amplitude errors, number of quantization bits, and bandwidth, and it can output two-dimensional beam patterns, sidelobe statistics, and measures of specialized interest. The program discussed here, called ARRSTATS, differs from those in Refs. 7 and 8 primarily in that it can model arrays with a hybrid phase- and time-steered architecture,[9] including strictly phased arrays and strictly time-steered arrays.[10] Also, it computes individual realizations of the hemispherical radiation pattern and obtains most measures of the array performance directly from the full pattern rather than from formulas or from pattern cuts chosen *a priori*. (Ref. 8 does not specify how its performance measures are obtained.) For example, the beam width cuts are always along the major and minor axes of the beam width ellipse, regardless of its orientation. Another special feature is a method for locating the beam peak that is highly accurate for nearly flat phase fronts; it is the only measure not obtained directly from the radiation pattern.

In more detail, the modeled array comprises a rectangular grid of phase-steered elements; these

Manuscript approved

are grouped into time-steered subarrays, which in turn are grouped into time-steered subapertures. The phases and times may be quantized. A random phase error may be associated with the elements, and random time errors may be assigned to the subarrays and subapertures. Also, random amplitude errors may be associated with the elements, subarrays, and subapertures. The user specifies the design frequency, at which the error-free times and phases would correctly steer the antenna, and the operating frequency, at which the array's behavior is simulated. The antenna may be steered to any direction. ARRSTATS assesses the array's performance by computing and analyzing the far-field radiation pattern or an ensemble of statistically identical patterns. It ignores polarization and mutual coupling between elements and assumes that the elements radiate uniformly into the forward hemisphere. For each computed pattern, it determines the following:

- the location and power density of the main beam peak
- the error in the main beam's location
- the main beam's angular limits, major and minor widths at half power, and orientation
- the directivity
- the ratios of powers in the main beam, sidelobes, and the radiation hemisphere
- the powers and distances of the sidelobes that are strongest and nearest to the main beam

Furthermore, it can track these measures as functions of a user-specified independent variable. ARRSTATS provides textual output of the performance measures, plots of radiation patterns, and plots of performance measures versus the independent variable.

ARRSTATS is a script written for version 5.3 of MATLAB, a commercial software package for technical computing.[11] Some elements of the program structure and some of the graphics facilities take advantage of features in version 5.3, but much of the code is compatible with earlier versions of MATLAB. ARRSTATS consists of one text file and is executed by typing its filename at the MATLAB command prompt. The program does not have an input user interface; instead, the user hardcodes input into the script before execution. These input points are tagged with the word "INPUT" in the code's comments.

The remainder of this report is structured as follows. Section 2 introduces the coordinate systems used for input and output, Section 3 explains the input parameters that describe the array, Section 4 specifies the model for the excitations, and Section 5 outlines the calculation of the radiation pattern. Section 6 describes the pattern measures, Section 7 discusses looping over multiple realizations and parameter values, and Section 8 exhibits the program's output. Two appendices are provided: Appendix A relates the variables used in this report and in the program, and Appendix B is a listing of the program code. Most of the report aims to describe aspects of the program's operation but does not give the details of the implementation or algorithms. The interested reader is directed to the code listing, specifically the comments that introduce each section of the program. Throughout this report, code variables are printed in a `monospaced` face, and braces ({}) enclose references to code line numbers except where the context suggests set notation.

2. COORDINATE SYSTEMS AND PROJECTIONS

ARRSTATS internally uses a three-dimensional Cartesian coordinate system to describe space. The array lies in the *x*-*y* plane and radiates into the half-space $z > 0$, as in Figure 1. (Although we describe a transmitter array, the case for a receiver array is identical.) The far-field spatial distribution of this radiation — that is, the radiation pattern — is a function of direction in the half-space. Two variables suffice to specify direction, and several pairs of variables are useful for this purpose. First, direction

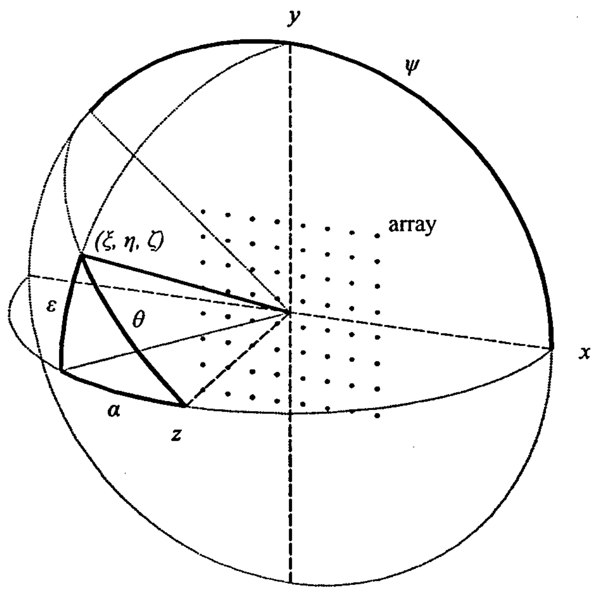

Figure 1 — Cartesian and spherical coordinates; θ, ψ, α, and ε are shown positive

cosines are the natural coordinates for calculating the array factor, as will be seen later. The direction cosines for a given direction are simply the Cartesian coordinates (ξ, η, ζ) of the corresponding unit vector. Specifying a direction in the half-space $z > 0$ requires only ξ and η; ζ may be obtained from the unit vector constraint if needed. Second, spherical coordinates are convenient for constructing flat projections of the radiation pattern, as it may be regarded as a function of location on a (curved) hemisphere. As shown in Figure 1, the polar angle θ for a given vector is the angle between the positive z axis and the vector, while the azimuth angle ψ is the angle between the positive x axis and the projection of the vector onto the x-y plane. Third, traditional antenna coordinates connect these simulations to an established context. Given the projection of a vector onto the x-z plane, the azimuth angle α is the angle between the projection and the z axis, while the elevation angle ε is the angle between the projection and the vector. These three sets of coordinates are related according to

$$\begin{aligned} \xi &= \sin\theta\cos\psi = -\cos\varepsilon\sin\alpha \\ \eta &= \sin\theta\sin\psi = \quad \sin\varepsilon \\ \zeta &= \cos\theta \qquad\;\; = \quad \cos\varepsilon\cos\alpha \end{aligned} \tag{1}$$

$$\begin{aligned} \cos\theta &= \quad \cos\varepsilon\cos\alpha = \zeta \\ \tan\psi &= -\tan\varepsilon/\sin\alpha = \eta/\xi \end{aligned} \tag{2}$$

$$\begin{aligned} -\tan\alpha &= \xi/\zeta = \tan\theta\cos\psi \\ \sin\varepsilon &= \eta \quad\; = \sin\theta\sin\psi. \end{aligned} \tag{3}$$

ArrStats also employs three projections of the hemisphere onto flat two-dimensional space: orthographic, Lambert azimuthal, and stereographic. The orthographic projection yields a view of the hemisphere from a particular vantage point without perspective distortion and is available for displaying the radiation pattern. The remaining projections both map the hemisphere to a disk such that boresight

corresponds to the center of the disk and grazing directions correspond to the perimeter of the disk. To describe these projections more specifically, we denote locations on the disk using polar coordinates (radius and angle). For both projections, the angular coordinate is set equal to the spherical azimuth angle ψ; the mapping from the spherical polar angle θ to the radius r distinguishes the two projections.

The Lambert projection preserves relative area: the ratio of two areas on the hemisphere equals that of the corresponding areas on the projected disk.[12] This property may be expressed by equating (to a proportionality constant) the spherical and planar surface areas:

$$\sin\theta\, d\theta\, d\psi = C r_L\, dr_L\, d\psi, \tag{4}$$

where the subscript "L" distinguishes the radius in the Lambert projection from that in the stereographic projection below. Canceling $d\psi$ and integrating both sides produces an integration constant, whose value and that of C are determined by the constraints that $r_L = 0$ when $\theta = 0$ and $r_L = 1$ when $\theta = \pi/2$. One finds

$$r_L = \sqrt{2}\sin\frac{\theta}{2}, \tag{5}$$

which deviates only slightly from a linear relationship for $\theta \in [0, \pi/2]$. When the radiation pattern is plotted with the Lambert projection, the areas occupied by structures such as the main beam and sidelobes are in true proportion to each other and to the total area.

While the stereographic projection does not preserve area, it is conformal.[12] The local scale is uniform in any direction; shape is preserved locally. Great and small circles on the hemisphere are projected into circles or straight lines, and the angle between two great circles on the hemisphere equals the angle at the intersection of their projected images. To derive the governing relationship, equate the aspect ratios of orthogonal derivatives, as

$$\frac{d\theta}{\sin\theta\, d\psi} = \frac{dr_S}{r_S\, d\psi}, \tag{6}$$

where the subscript "S" denotes the stereographic projection. Canceling and integrating as before yields

$$r_S = \tan\frac{\theta}{2}. \tag{7}$$

ARRSTATS internally uses the stereographic projection when identifying the major and minor axes of the beamwidth ellipse (see Section 6.3) and makes it available for plotting the radiation pattern.

3. ARRAY PARAMETERS

Several parameters describe the antenna's geometry and associated frequencies {29–94}. As Figure 2 illustrates, the antenna elements occupy a regular rectangular grid in the x-y plane; the element spacings in each dimension, d_x and d_y, may differ. The elements are grouped hierarchically at three levels. The array is subdivided into L_x by L_y subapertures, each of which has an associated time delay for steering. Descending to the next level, each subaperture contains M_x by M_y subarrays, each of which likewise has a steering time delay. Finally, each subarray contains N_x by N_y elements, each of which is phase-steered. Regular element spacing is maintained across subarray and subaperture boundaries. To

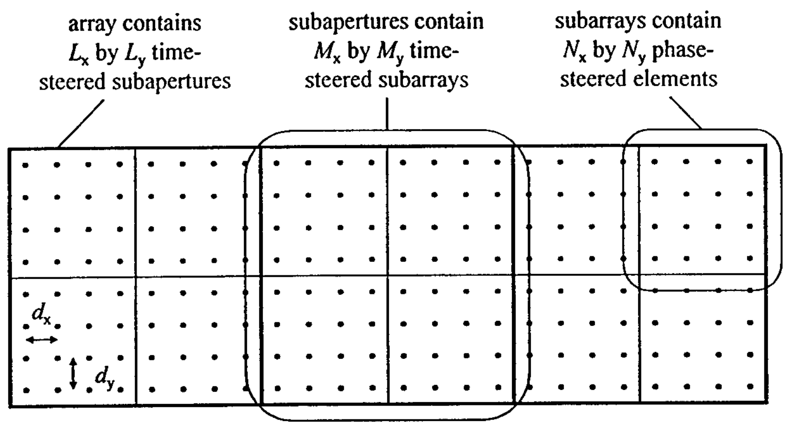

Figure 2 — Array structure and geometry

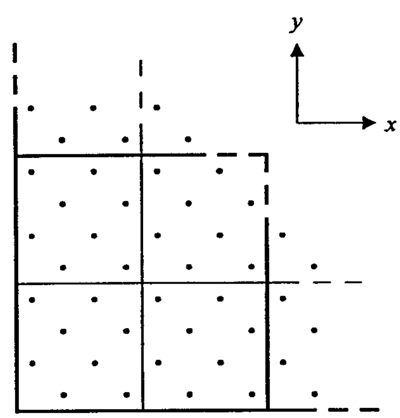

Figure 3 — Element layout when `diamond` flag is nonzero

simulate an array with one time-steered level and one phase-steered level, L_x and L_y should be set to 1; for a strictly phased array, also set M_x and M_y to 1. On the other hand, one may model a strictly time-steered array with one or two hierarchical levels by setting N_x and N_y to 1 and $\delta\varphi$ to 360° (this renders the phase shifters ineffective; see Section 4.1).

Two additional program parameters specify special antenna geometries. If the flag `diamond` is nonzero, elements are effectively removed from even diagonals (the zeroth diagonal originates at the lower-left element), leaving a diamond pattern of active elements as in Figure 3. The parameter `azmthOffst` rotates the antenna about the z axis (boresight); it is the angle of the positive x axis of the array above the azimuth reference. Within ArrStats, all calculations are performed in the antenna coordinate system; the steering vector provided by the user is transformed to the antenna coordinate system before processing, and output coordinates and plotted structures are transformed from the antenna coordinate system after processing.

Finally, two parameters supply frequency information. The reference or design frequency f_0 is that at which the time and phase delays would correctly steer the antenna in the absence of error and quantization. The operating frequency f is that at which the simulation is to be performed. Multiple frequencies may be considered sequentially as described in Section 7.

4. EXCITATIONS

4.1. Error-Free Excitations

We now construct the element excitations in detail to show the structure of the array model, beginning with the error-free case {338–410}. Because only one frequency is considered at a time, time delays may be expressed as equivalent phase delays. We therefore decompose the excitations into magnitude and phase as

$$\bar{a}_{n_x n_y} = |\bar{a}_{n_x n_y}| \exp(-i\bar{\Phi}_{n_x n_y}), \quad n_w \in \{0,1,\ldots,L_w M_w N_w - 1\}, \quad w \in \{\mathrm{x}, \mathrm{y}\}, \tag{8}$$

where the overbars indicate the error-free case and the n_w label the elements across the face of the array, ignoring subaperture and subarray boundaries. The error-free magnitudes are made equal for all active elements and normalized to unit total power, so that

$$\sum_{n_x} \sum_{n_y} |\bar{a}_{n_x n_y}|^2 = 1. \tag{9}$$

The phases are derived from the condition that at the reference frequency the far-field radiation must interfere perfectly constructively in the direction of the steering vector. This implies that the phase must progress linearly across the face of the array as

$$\bar{\Phi}_{n_x n_y} = -k_0 (d_x s_x n_x + d_y s_y n_y) + \text{const.}, \tag{10}$$

where $k_0 = 2\pi f_0 / c$ is the reference wave number and (s_x, s_y) are the direction cosines of the steering vector {96–166}. Considering the architecture of the array, the phases must be built up from the equivalent time delays at the subaperture and subarray levels and the phase delays at the element levels. Based on Eq. (10), the phase step in direction w between adjacent elements in a subarray must be

$$\Delta\varphi_w = -k_0 d_w s_w. \tag{11}$$

Likewise, since each subarray contains N_x by N_y elements, adjacent subarrays within a subaperture must have a phase difference of $N_w \Delta\varphi_w$, which is equivalent to a time step

$$\begin{aligned} \Delta t_w &= \frac{1}{\omega_0} N_w \Delta\varphi_w \\ &= -\frac{1}{c} N_w d_w s_w, \end{aligned} \tag{12}$$

where $\omega_0 = 2\pi f_0$ is the reference angular frequency, and the time step across subapertures follows similarly as

$$\Delta T_w = -\frac{1}{c} N_w M_w d_w s_w. \tag{13}$$

In practical arrays, the time and phase delays are often quantized, leading to violations of Eq. (10) for general steering angles. We suppose that the beamformer is capable of mitigating the effects of subaperture and subarray quantization by adjusting the subarray and element delays. That is, the delay lost or gained in each subaperture due to quantization can be balanced by additional or reduced delay in the subarrays, assuming that the quantization interval of the subarrays is less than that of the subapertures. Likewise, the error due to subarray quantization can be balanced by adjusting the element phases, subject to a similar assumption. To exhibit this scheme mathematically, we introduce the subaperture, subarray, and element quantization intervals δT, δt, and $\delta\varphi$, respectively. We also introduce subaperture labels l_x and l_y and subarray labels m_x and m_y; as the n_w ignore subaperture and subarray boundaries, so the m_w ignore subarray boundaries. More specifically, we obtain the l_w and m_w from the n_w according to

$$\begin{aligned} l_w &= \left\lfloor \frac{n_w}{N_w M_w} \right\rfloor, \quad l_w \in \{0, 1, \ldots L_w - 1\}, \quad \text{and} \\ m_w &= \left\lfloor \frac{n_w}{N_w} \right\rfloor, \quad m_w \in \{0, 1, \ldots L_w M_w - 1\}, \end{aligned} \tag{14}$$

where $\lfloor x \rfloor$ is the greatest integer not exceeding x.

We define the subaperture time delays without quantization or error to be

$$\overline{T}_{n_x n_y} = l_x \Delta T_x + l_y \Delta T_y, \tag{15}$$

where the l_w are implicitly dependent upon the n_w, and the quantized subaperture time delays are then

$$\hat{T}_{n_x n_y} = \delta T \operatorname{round}(\overline{T}_{n_x n_y} / \delta T), \tag{16}$$

where round(x) is the integer nearest x. The subarray time delays contain an additional term that compensates for the subaperture quantization:

$$\begin{aligned} \bar{t}_{n_x n_y} &= m_x \Delta t_x + m_y \Delta t_y - \hat{T}_{n_x n_y}, \\ \hat{t}_{n_x n_y} &= \delta t \operatorname{round}(\bar{t}_{n_x n_y} / \delta t), \end{aligned} \tag{17}$$

where the m_w are implicitly dependent upon the n_w, and the element phase delays contain two similar terms:

$$\begin{aligned} \overline{\varphi}_{n_x n_y} &= n_x \Delta\varphi_x + n_y \Delta\varphi_y - \omega_0 (\hat{T}_{n_x n_y} + \hat{t}_{n_x n_y}), \\ \hat{\varphi}_{n_x n_y} &= \delta\varphi \operatorname{round}(\overline{\varphi}_{n_x n_y} / \delta\varphi) \end{aligned} \tag{18}$$

In so defining the delays, we have implicitly chosen the constant in Eq. (10) to be zero. This choice implies that the lowest and leftmost components (those with $l_w = 0$, $m_w = 0$, or $n_w = 0$) have no associated delay regardless of the steering vector, whereas the highest and rightmost components (having l_w =

$L_w - 1$, $m_w \bmod M_w = M_w - 1$, or $n_w \bmod N_w = N_w - 1$) have delays that depend strongly on the steering vector.

In the program, quantization may be avoided by setting the quantization steps to zero. The above equations are then equivalent to

$$\begin{aligned} \overline{T}_{n_x n_y} &= \hat{T}_{n_x n_y} = l_x \Delta T_x + l_y \Delta T_y, \\ \overline{t}_{n_x n_y} &= \hat{t}_{n_x n_y} = (m_x \bmod M_x)\Delta t_x + (m_y \bmod M_y)\Delta t_y, \quad \text{and} \\ \overline{\varphi}_{n_x n_y} &= \hat{\varphi}_{n_x n_y} = (n_x \bmod N_x)\Delta\varphi_x + (n_y \bmod N_y)\Delta\varphi_y. \end{aligned} \tag{19}$$

We note that $m_w \bmod M_w$ is the index of subarray m_w within its parent subaperture, and likewise $n_w \bmod N_w$ is the index of element n_w within its parent subarray.

For each element, the net (possibly quantized) phase at the operating frequency is the sum of equivalent phase contributions from the three hierarchical levels:

$$\hat{\Phi}_{n_x n_y} = \omega(\hat{T}_{n_x n_y} + \hat{t}_{n_x n_y}) + \hat{\varphi}_{n_x n_y}, \tag{20}$$

where $\omega = 2\pi f$ is the operating angular frequency. This quantized phase assumes the place of $\Phi_{n_x n_y}$ in Eq. (8). At the reference frequency ($\omega = \omega_0$) and with no phase quantization, this construction of the phase yields the linear progression of Eq. (10).

4.2. Erroneous Excitations

We model the errors in a real antenna by applying amplitude and phase errors to each level of the antenna hierarchy {525–549}. Amplitude errors multiply the error-free amplitudes by factors of the form $1 + \rho$ where ρ is a random number, distributed normally with zero mean. Each level contributes such errors, so that the erroneous amplitude for element (n_x, n_y) is

$$|a_{n_x n_y}| = |\overline{a}_{n_x n_y}|(1 + R_{l_x l_y})(1 + r_{m_x m_y})(1 + \rho_{n_x n_y}), \tag{21}$$

where R, r, and ρ correspond to subapertures, subarrays, and elements, respectively. This model allows corresponding elements in different subarrays to contribute distinct errors, and likewise for corresponding subarrays within different subapertures. The user specifies the standard deviations σ_R, σ_r, and σ_ρ of the respective amplitude errors.

Time and phase errors add to the error-free (but possibly quantized) time and phase delays. The subapertures and subarrays contribute random time errors $\widetilde{T}$ and $\widetilde{t}$, respectively, and the elements contribute random phase errors $\widetilde{\varphi}$, all drawn from zero-mean normal distributions. The user specifies the corresponding standard deviations σ_T, σ_t, and σ_φ. Additionally, the user may specify a deterministic time error $\widetilde{\mathcal{T}}$ for each subaperture. The net equivalent phase error for element (n_x, n_y) is

$$\widetilde{\Phi}_{n_x n_y} = \omega(\widetilde{\mathcal{T}}_{l_x l_y} + \widetilde{T}_{l_x l_y} + \widetilde{t}_{m_x m_y}) + \widetilde{\varphi}_{n_x n_y}, \tag{22}$$

and the total erroneous phase is the sum of the quantized and error phases:

$$\Phi_{n_x n_y} = \hat{\Phi}_{n_x n_y} + \tilde{\Phi}_{n_x n_y} . \tag{23}$$

Finally, the erroneous complex excitations are

$$a_{n_x n_y} = |a_{n_x n_y}| \exp(-i\Phi_{n_x n_y}) . \tag{24}$$

5. RADIATION PATTERN

In standard array theory with mutual coupling ignored, the field pattern is the product of the array factor and the element factor. In ARRSTATS, the element factor is unity, corresponding to uniform hemispherical radiation, so the field pattern equals the array factor. (See the code {2595–2715} for notes on expanding ARRSTATS.) Given the complex excitations, the array factor (and field pattern) in the direction (ξ, η) is given by the two-dimensional Fourier transform {551–561}

$$g(\xi,\eta) = \sum_{n_x} \sum_{n_y} \exp[-ik(\xi d_x n_x + \eta d_y n_y)] a_{n_x n_y} , \tag{25}$$

where $k = 2\pi f / c$ is the operating wave number. ARRSTATS uses a fast Fourier transform to obtain g in discrete directions given by

$$\begin{aligned} \xi_{q_x} &= \frac{2\pi q_x}{k d_x Q_x}, \quad q_x \in \{0,1,\ldots,Q_x - 1\}, \quad \text{and} \\ \eta_{q_y} &= \frac{2\pi q_y}{k d_y Q_y}, \quad q_y \in \{0,1,\ldots,Q_y - 1\}, \end{aligned} \tag{26}$$

where Q_x and Q_y are the number of points in the transform in each dimension {168–173}. The (ξ_{q_x}, η_{q_y}) grid is extrapolated to all of visible space using {412–497, 558}

$$\begin{aligned} g(\xi_{q_x+Q_x}, \eta_{q_y}) &= g(\xi_{q_x}, \eta_{q_y}) \quad \text{and} \\ g(\xi_{q_x}, \eta_{q_y+Q_y}) &= g(\xi_{q_x}, \eta_{q_y}), \end{aligned} \tag{27}$$

which are valid for all integers q_x and q_y. Because of the normalization condition of Eq. (9),

$$|g(\xi,\eta)| \le 1 \tag{28}$$

for all ξ and η, with the equality holding only where the excitations interfere perfectly constructively. Therefore, when the field pattern is expressed in decibels, 0 dB corresponds to perfectly constructive interference.

6. PATTERN MEASURES

The code at the heart of the program analyzes the radiation pattern to obtain several measures that quantify the characteristics and performance of the array. The following subsections describe these measures, generally focusing on the concepts behind them and on their interpretation rather than on the specific method of calculation. Details of the algorithms and further discussion may be found in the code.

6.1. Pointing Vector, Pointing Error, and Peak Power Density

One of the most significant pattern measures is the direction of maximum radiation, here called the pointing vector. The program determines it directly from the field pattern and also indirectly from the excitations; the final pointing vector is a weighted average of the two, as discussed below. In identifying the pointing vector from the field pattern {563–787}, the program first locates the pattern's maximum magnitude over the grid of discrete direction cosines. For a well-formed beam, the neighboring samples should fall off parabolically, so they are fitted to the elliptic paraboloid[13]

$$|g(\xi,\eta)| = \frac{1}{2}U\xi^2 + W\xi\eta + \frac{1}{2}V\eta^2 + X\xi + Y\eta + Z \tag{29}$$

in a least-squares sense. If $UV > W^2$, as should be the case for a normal beam, the conic is indeed elliptical (corresponding to its level curves), and its maximum occurs at $(\xi, \eta) = (p_{1x}, p_{1y})$, where p_{1x} and p_{1y} satisfy

$$\begin{pmatrix} U & W \\ W & V \end{pmatrix}\begin{pmatrix} p_{1x} \\ p_{1y} \end{pmatrix} + \begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}. \tag{30}$$

The coordinate pair (p_{1x}, p_{1y}) is taken to be the pointing vector for the first method. The deviation of the pattern samples from conic form is used to construct a covariance ellipse for p_{1x} and p_{1y} that expresses the uncertainty in their values.

The second method determines the pointing vector from the complex excitations {789–961}. Because a well-behaved array will have a nearly flat phase front, the excitation phases $\Phi_{n_x n_y}$ are fit to the plane $-kn_x d_x \xi - kn_y d_y \eta - \Phi_0$, weighted according to the excitation magnitudes. The pointing vector coordinates (p_{2x}, p_{2y}) are the direction cosines (ξ, η) that give the best fit. As with the first method, a covariance ellipse for p_{2x} and p_{2y} is obtained from a measure of the deviation from the plane.

Each method is useful but limited. The first, the fit of the transform, is robust even for poorly aimed and malformed beams, but it is limited by the resolution of the Fourier transform. The second, the fit of the excitations, is independent of transform resolution but accurate only for nearly planar phase fronts, approaching the exact solution as the phase and time errors and quantization intervals decrease. To obtain a single pointing vector (p_x, p_y), the program averages pointing vectors from the two methods, weighting each by the inverse of the area of its covariance ellipse {963–990}. An average covariance ellipse is also constructed in a consistent manner.

With the final pointing vector in hand, the pointing error γ is straightforwardly obtained {992–1046} from

$$\begin{aligned} \cos\gamma &= \mathbf{s}\cdot\mathbf{p} \\ &= s_x p_x + s_y p_y + s_z p_z, \end{aligned} \tag{31}$$

where s_z and p_z follow from unit magnitude constraints on $\mathbf{s}$ and $\mathbf{p}$. If one desires the uncertainty in γ due to uncertainty in $\mathbf{p}$, an alternate calculation based on

$$\sin\gamma = |\mathbf{s}\times\mathbf{p}| \tag{32}$$

may be selected in the program.

The peak power density

$$P_{\max} = |\, g(p_x, p_y)|^2 \tag{33}$$

is calculated directly from Eq. (25).

6.2. Main Beam Region

The main beam region is the set of field pattern samples that are judged to belong to the main beam. Although normally not of interest as a final measure, it is essential for obtaining subsequent measures. It may be constructed conceptually by imagining a contour at an adjustable level. Beginning at the pattern maximum, we decrease the level so that the contour expands in size, following the topology of the main beam. Eventually the contour will intersect a local minimum or a saddle point; the closed contour about the pattern maximum at that level delineates the main beam region inside from the sidelobe region outside. Equivalently, that contour is the lowest one containing the global maximum that encircles no other local maxima. The power level of the contour is called the beam depth. Generally, well-formed beams are deep (that is, the beam depth is much less than one), while malformed beams are shallow, but the user should keep in mind that the beam depth depends on the transform resolution. In the program {1048–1137}, the beam depth is stored in the variable `beamDepthDB` in decibels relative to $P_{\max}$ and is output to the user. Information obtained while determining the main beam region is used in finding the main beam width and roll, below, and the region itself is used directly in calculating the powers radiated into the main beam and sidelobes.

6.3. Main Beam Width and Roll

The level contours of the main beam generated by a two-dimensional array are nominally ellipses;[14,15] therefore they can be described by their center location, major and minor axes, and orientation. Having already obtained a measure of the beam's location in the form of the pointing vector, we use the major and minor axes and orientation of the elliptical contour at a given power level to describe the beam's shape {1139–1403}. The conventional power level is $P_{\max}/2$, or about 3 dB down. The angle subtended by the ellipse's longest diameter — its major axis — is taken as the beam's major width (that is, full width at half maximum power); that subtended by its shortest diameter, the beam's minor width. The ellipse orientation gives the roll angle, but we must choose an origin for the orientation angle. Construct three great circles as in Figure 4: *A*, along the azimuth reference; *B*, connecting boresight and the pointing vector; and *C*, along the beam's major width. The roll angle is defined as the sum of the angle between *A* and *B* and that between *B* and *C*. It happens that the roll angle so defined is merely the orientation of the major width relative to the azimuth reference when viewed in the stereographic projection, which preserves angles between great circles. Moreover, because the great circles along the beam's major and minor widths intersect orthogonally on the hemisphere, their stereographic projections do also. These facts motivate the program's use of stereographic coordinates for fitting an ellipse to the level contour and determining its major and minor axes and orientation. However, the roll angle and beam widths thereby obtained are approximate for two reasons. First, for beams off boresight, the great circles along the major and minor widths project as circles, whereas the major and minor axes of the projected ellipse are straight line segments. Second, the local scale in the projection increases away from boresight, artificially enlarging the portion of the beam farthest from boresight.[12] The error may become significant for beams far from boresight. A more sophisticated method of determining the beam widths is suggested near the end of the program code.

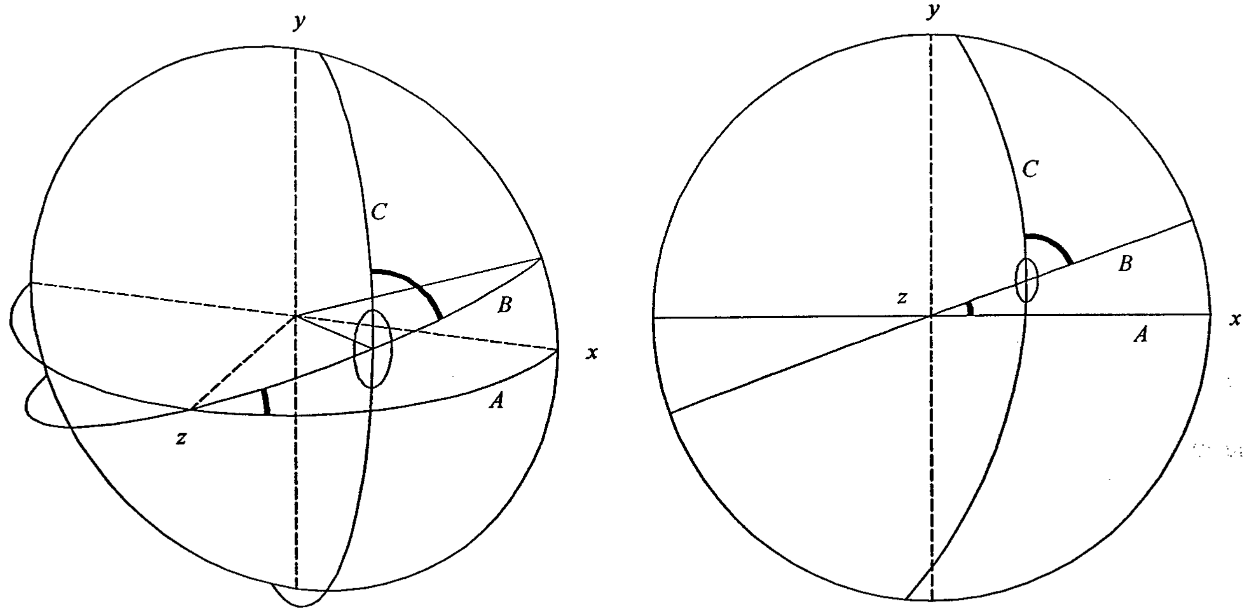

Figure 4 — Beam width ellipse and roll angle. The great circles A, B, and C are described in the text, and the angles between them, indicated with thick arcs, are added to obtain the roll angle. The left projection is orthographic; the right, stereographic. The spherical coordinates of the pointing vector are $\theta = 40°$ and $\psi = 20°$, the beam's major and minor widths are 16° and 8°, and the roll angle is 90°.

6.4. Directivity, Power Ratios, and Average Sidelobe Level

The next measures obtained all depend on integrals of the power pattern over solid angle regions {1405–1527}. The integrals are calculated from the discrete samples of the pattern using the midpoint approximation, as detailed in the code. The solid angle regions of interest are visible space (the half-space $z > 0$); the main beam, as described by the main beam region, above; and the sidelobes, defined as all regions of visible space not in the main beam. The program determines the total powers radiated into these regions; call them Π_v for visible space, Π_m for the main beam, and Π_s for the side lobes. We have

$$\Pi_v = \Pi_m + \Pi_s . \tag{34}$$

The directivity is the ratio of maximum to average power densities, assuming no back radiation into $z < 0$:

$$D = \frac{P_{max}}{\Pi_v / 4\pi} . \tag{35}$$

The program also calculates the power ratios Π_m / Π_v, Π_m / Π_s, and Π_v / Π_s, which generally decrease as the beam degrades. Finally, the average sidelobe level relative to the beam peak is

$$L_{avg} = \frac{\Pi_s}{\Omega_s P_{max}} , \tag{36}$$

where Ω_s is the solid angle occupied by the sidelobes.

6.5. Powers and Distances of Largest and Nearest Sidelobes

The last analysis identifies the sidelobe with the largest power density and the sidelobe closest to the main beam, and for each it reports the power density and angular distance from the main beam {1529–1677}. The power levels and locations of the sidelobes are obtained by fitting to elliptic paraboloids (Eq. (29)), and the power levels are reported relative to the beam peak P_{max}.

7. MULTIPLE REALIZATIONS AND PARAMETER VALUES

The analyses described above apply to the radiation pattern corresponding to a single set of array parameters and one realization of any random variables. ARRSTATS contains two outer loops with which it analyzes multiple radiation patterns; one is a loop over realizations of random variables, and the other loops over a user-selected independent variable. The loop over realizations {190–218, 521–523, 1679–1819, 1905} is motivated by the following: When simulating random errors, one is usually interested not in the performance obtained by one realization of the errors but rather in the performance statistics for an ensemble of statistically identical arrays. To that end, ARRSTATS can generate an arbitrary, user-specified number of realizations for which it will accumulate performance statistics. The program outputs the mean and standard deviation of each performance measure. The second loop {175–188, 326–336, 2578–2579}, over an independent variable, allows the user to examine the variation of performance measures as the variable changes. Possible independent variables include but are not limited to the operating and reference frequencies, steering angles, error parameters, quantization intervals, and even parameters of the array geometry. ARRSTATS produces a graph showing each measure as a function of the independent variable, as discussed below.

8. PROGRAM OUTPUT

ARRSTATS outputs its results in three ways: textual output of running statistics, a plot of the radiation pattern for the last realization in an ensemble, and a summary plot of the performance measures as functions of the independent variable. The textual output is a table like that in Fig. 5 printed to the MATLAB command window {1864–1903}. The values in the table are the means and standard deviations of the performance measures for all members of the statistical ensemble that have been realized thus far. The table may be interpreted according to the descriptions in Section 6, keeping in mind the following.

```
Means and [std devs] for 16 of 16 realizations
beam direction : (29.954, 60.007) deg, std dev 0.038 deg
pointing error :   0.0528 [0.0279] deg
peak power dens:  -0.426  [0.034 ] dB
beam depth     : -24.66   [1.49  ] dB re peak
beam width     : ( 2.095  [0.005 ], 1.813 [0.004]) deg
beam roll      :  59.56   [0.65  ] deg
directivity    :  38.894  [0.034 ] dB
power ratio m/v:  -1.360  [0.030 ] dB
power ratio m/s:   4.347  [0.113 ] dB
power ratio v/s:   5.707  [0.083 ] dB
avg sidelobe   : -41.61   [0.11  ] dB re peak
nrst sidelobe  : -13.16   [0.69] dB re peak, 3.00 [0.01] deg off beam
lgst sidelobe  : -12.30   [0.58] dB re peak, 3.11 [0.09] deg off beam
```

Figure 5 — Textual output of running statistics. The parameters of this example are in the code listing; for the values above, the standard deviation of the subarray time error is 10 ps (`stdTimeMPS` = 10).

The coordinates specifying the beam direction are the spherical coordinates (θ and ψ) of the average pointing vector {1821–1862}. The standard deviation of the beam direction is an estimate of the rms angular deviation of the ensemble of pointing vectors from their mean {1821–1862}. Both the beam direction coordinates and roll angle include compensation for the azimuth offset `azmthOffst`. The peak power density is relative to perfectly constructive interference, and the beam depth, average sidelobe level, and levels of the nearest and largest sidelobes are relative to the peak power density (suggested by the use of "dB re peak" in the table).

To aid in visualizing an array's performance, ARRSTATS can graphically present the radiation pattern and several of the performance measures as in Fig. 6 {220–324, 2138–2574}. A significant

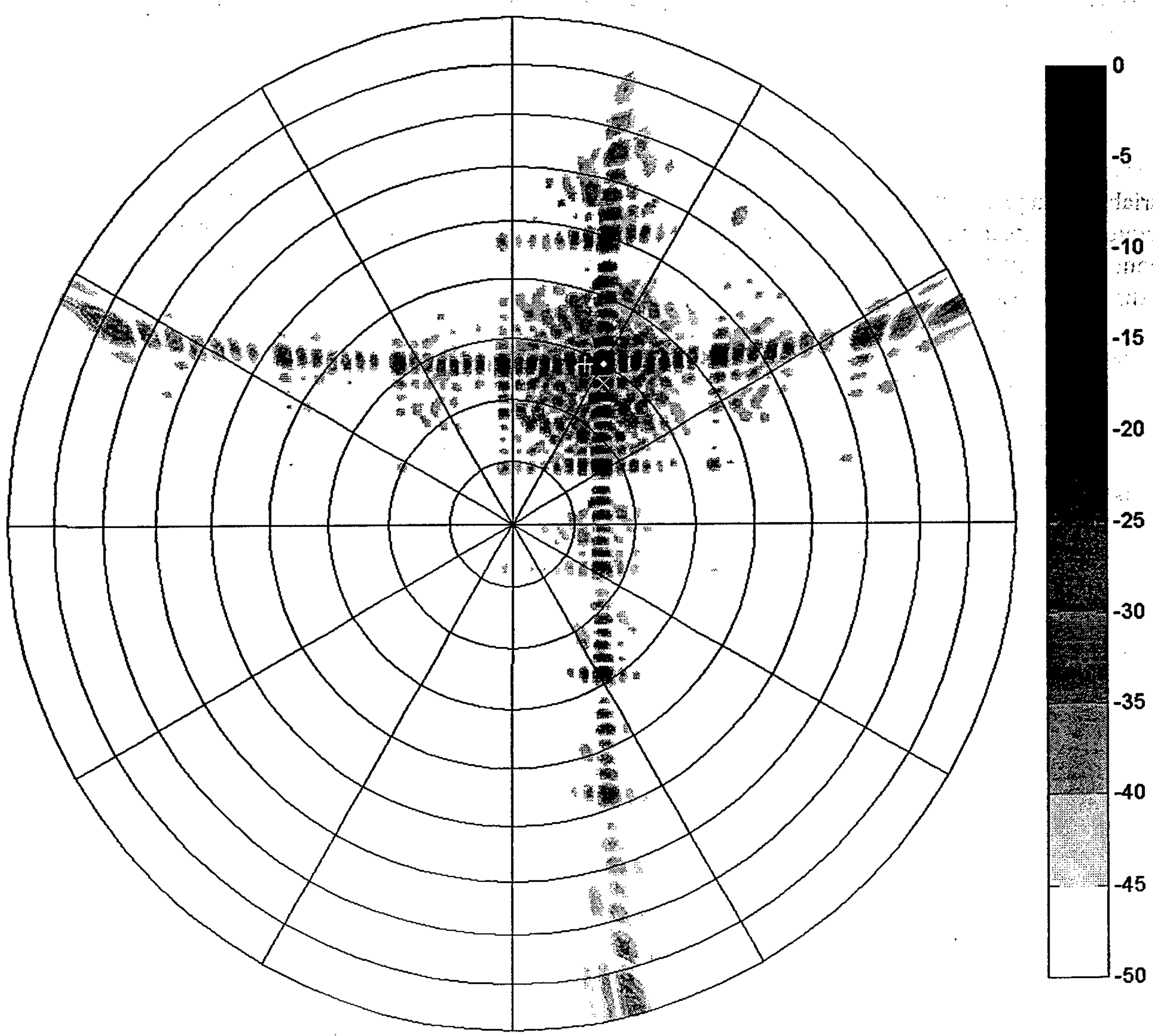

Figure 6 — Lambert projection of the radiation pattern with a spherical coordinate grid superimposed. Boresight is at the center, and the steering vector is ($\theta = 30°$, $\psi = 60°$). The white dot denotes the beam peak; ×, the largest sidelobe; and +, the nearest sidelobe. The legend gives the power in dB relative to perfectly constructive interference. The pattern is one realization of the case `stdTimeMPS` = 10 for the parameters given in the code listing.

plotting option is the choice of projection from among those described in Section 2. Briefly, the Lambert projection preserves the relative areas of regions (for example, the size of the main beam relative to the hemisphere or to prominent sidelobes), the stereographic projection preserves local shape (and the orthogonality of the major and minor beam width cuts), and the orthographic projection gives a picture of the hemisphere. The pattern may be plotted linearly or logarithmically in power, and the logarithmic depth may be specified. The user may also specify the color map and shading method to use. The performance measures that can be graphically indicated on the radiation pattern include the pointing vector, the axes of the uncertainty ellipse of the pointing vector, the actual and fitted beam width contours, the main beam region, the locations of the nearest and largest sidelobes, and other measures of less frequent interest. If multiple realizations are generated, the radiation pattern will be plotted only for the last realization as a representative of the ensemble.

If the outer loop over an independent variable is used, the summary figure plots the performance measures as functions of the independent variable, as in Fig. 7 {220–324, 1907–2136}. The figure groups fifteen measures (all except the beam direction) into eight subplots and utilizes distinct colors or line styles and both left and right axes for the ordinates. The title of each subplot gives the names of the measures; where multiple colors or linestyles appear, each measure name is followed by a color or style code in parenthesis, and where left and right axes are used, an axis code ("L" or "R") also appears. If more than one realization was generated for each value of the independent variable, error bars extend one standard deviation above and one standard deviation below each point.

9. SUMMARY

We have introduced a computer program for assessing the effects of errors in rectangular-grid planar array antennas. The most general array comprises phase-steered elements grouped into time-steered subapertures and subarrays; this includes strictly phase-steered arrays and time-steered arrays as particular cases. The time and phase delays may be quantized, and random errors may be assigned to the times, phases, and amplitudes. From simulated radiation patterns, the program obtains performance measures over statistical ensembles and as functions of a user-specified independent variable, producing textual and graphical output.

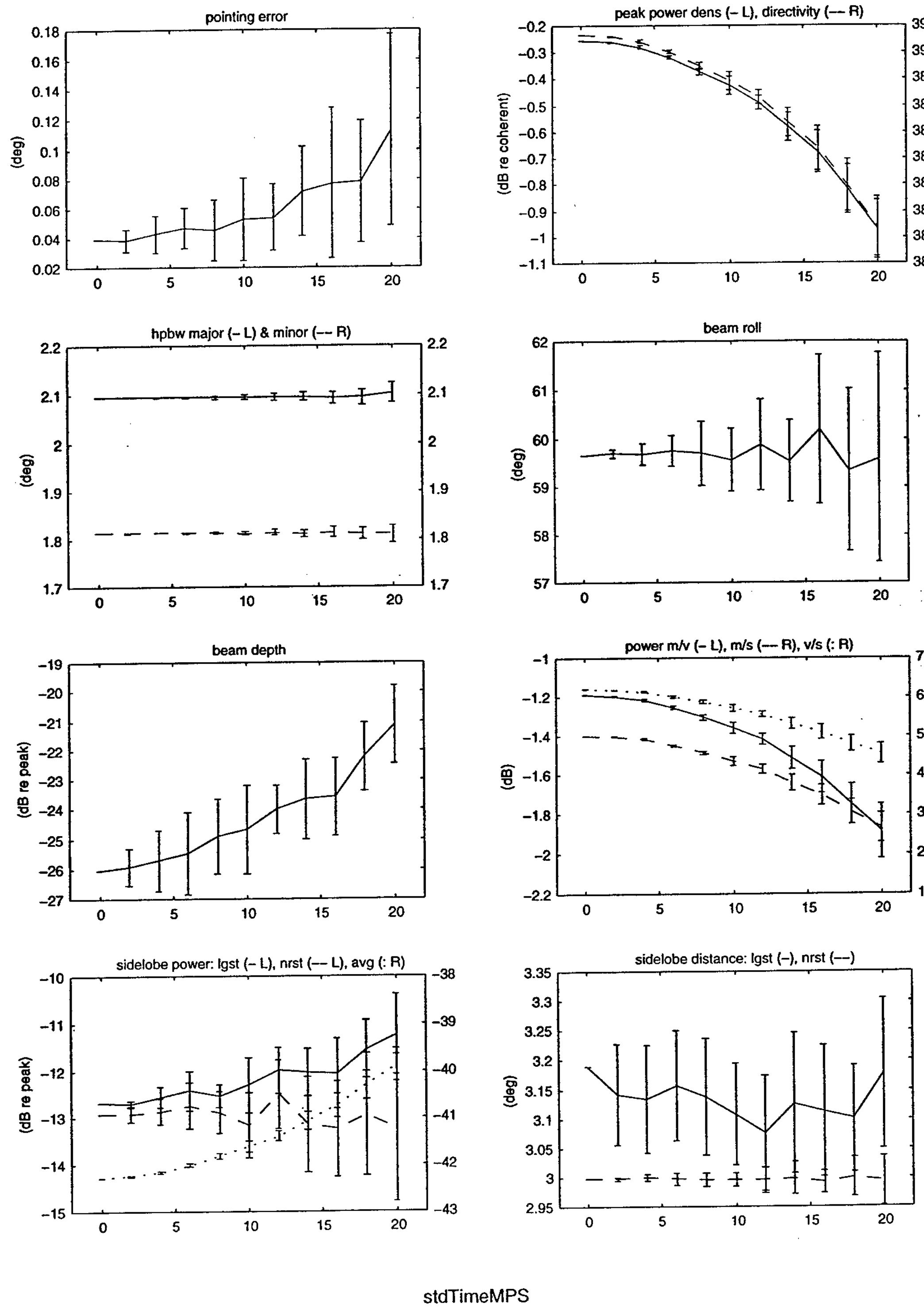

Figure 7 — Summary plots. This example shows the degradation of performance as the standard deviation of the subarray time error (`stdTimeMPS`, in picoseconds) increases. The parameters of the run may be found in the code listing.

Appendix A

LIST OF VARIABLES

This list of significant variables mentioned in this report gives their symbol in this report, coded name, and description. It does not include all program variables. Ellipses (...) stand for prefixes, and asterisks (*) indicate that the variable holds the specified quantity temporarily.

Text	Code	Description
x, y, z		coordinates in real space
ξ, η, ζ	`dirCosX`, ...`Y`, ...`Z`	direction cosines
s_x, s_y	`sx`, `sy`	steering vector direction cosines
θ	...`Polar`	spherical polar coordinate
ψ	...`Azmth`	spherical azimuth coordinate
α	...`Az`	traditional azimuth coordinate
ε	...`El`	traditional elevation coordinate
r		radial coordinate of projection
d_x, d_y	`dx`, `dy`	element spacings
L_x, L_y	`numLX`, `numLY`	numbers of subapertures
M_x, M_y	`numMX`, `numMY`	numbers of subarrays per subaperture
N_x, N_y	`numNX`, `numNY`	numbers of elements per subarray
l_x, l_y	`lx`, `ly`	subaperture labels
m_x, m_y	`mx`, `my`	subarray labels
n_x, n_y	`nx`, `ny`	element labels
	`diamond`	indicates diamond element pattern
	`azmthOffst`	antenna rotation angle about boresight
f_0	`fRef`	reference frequency
f	`fOpr`	operating frequency
k_0		reference wave number
k		operating wave number
ω_0		reference angular frequency
ω		operating angular frequency
σ_R	`stdAmplL`	standard deviation of subaperture amplitude error
σ_r	`stdAmplM`	standard deviation of subarray amplitude error
σ_ρ	`stdAmplN`	standard deviation of element amplitude error
R		random subaperture amplitude error
r		random subarray amplitude error

Text	Code	Description
ρ		random element amplitude error
$\bar{a}$	`excMagIdl`	error-free (ideal) excitation magnitude
a	`excMag`	erroneous (actual) excitation magnitude
δT	`qntTimeL`	subaperture time quantization interval
δt	`qntTimeM`	subarray time quantization interval
$\delta\varphi$	`qntPhseN`	element phase quantization interval
$\Delta T_x, \Delta T_y$		subaperture time steps
$\Delta t_x, \Delta t_y$		subarray time steps
$\Delta\varphi_x, \Delta\varphi_y$		element phase steps
$T_{n_x n_y}$	`TimeL*`	error-free subaperture time delays
$t_{n_x n_y}$	`TimeM*`	error-free subarray time delays
$\varphi_{n_x n_y}$	`PhseN*`	error-free element phase delays
$\hat{T}_{n_x n_y}$	`TimeL`	quantized subaperture time delays
$\hat{t}_{n_x n_y}$	`TimeM`	quantized subarray time delays
$\hat{\varphi}_{n_x n_y}$	`PhseN`	quantized element phase delays
σ_T	`stdTimeL`	standard deviation of subaperture time error
σ_t	`stdTimeM`	standard deviation of subarray time error
σ_φ	`stdPhseN`	standard deviation of element phase error
$\tilde{\mathscr{T}}$	`ofsTimeL`	deterministic subaperture time error
$\tilde{T}$		random subaperture time error
$\tilde{t}$		random subarray time error
$\tilde{\varphi}$		random element phase error
$\bar{\Phi}$		error-free excitation phase
$\hat{\Phi}$	`excPhsIdl`	quantized excitation phase
$\tilde{\Phi}$	`excPhsErr`	excitation phase error
Φ	`excPhs`	erroneous (actual) excitation phase
g	`g`	field pattern
Q_x, Q_y	`tx, ty`	number of points in Fourier transform
p_x, p_y	`px, py`	pointing vector direction cosines
γ	`errPoint`	pointing error
P_{max}	`gSqrMax`	maximum power density
L_{avg}	`powerSideAvgDB`	average relative sidelobe level (`powerSideAvgDB` $= 10 \log_{10} L_{avg}$)
Π_v	`powerVisb`	power radiated into visible space
Π_m	`powerMain`	power radiated into the main beam
Π_s	`powerSide`	power radiated into the side lobes
D	`directivityDB`	directivity (`directivityDB` $= 10 \log_{10} D$)

Appendix B

PROGRAM LISTING

```
% Calculate performance parameters of an array with excitation errors
%
% The excitation time and phase convention is exp (-i (omega t + phi));
% positive (negative) phases phi correspond to a leading (lagging)
% excitation.  Distances are stored in meters; times, nanoseconds;
% frequencies, gigahertz.  Angles (both geometric and phase) are always
% specified in radians.  Generally, the coordinate x increases with the
% column index; y, with the row index.
%
% Parameters that may be changed by the user are marked "INPUT" in
% comments.  Frequently-used inputs appear near the top of the program,
% but some inputs are defined elsewhere, particularly in the plotting
% section.
%
% Suggestions for improvements are provided at the end of the program.
%
% This program is coded for Matlab version 5.3 (Release 11), although
% most of the code will run under version 4.2.
%
% Written by Stan West, 1998, 1999
% U.S. Government work not subject to copyright

% Declare physical and conversion constants
%
c = 0.299792458;   % speed of light in m/ns
rpd = pi / 180;    % radians per degree
twopi = 2 * pi;

% Set operating frequency relative to reference frequency
%
% At the reference frequency, or fOpr = fRef, the time and phase delays
% will properly steer the antenna in the absence of error and
% quantization.
%
fRef =  3.0;                  % INPUT reference frequency in GHz
fOpr = fRef + 1 * 0.5 / 2;    % INPUT operating frequency in GHz
% center frequency + [-1 ... 1] * bandwidth / 2

% Describe array geometry and error statistics
%
% Elements lie on a regular planar grid and are grouped heirarchically.
% The array is subdivided into numLX subapertures in the x dimension and
% numLY subapertures in the y dimension.  Each subaperture is
% time-steered and has associated with it a user-set deterministic
% absolute time error, a random absolute time error, and a random
% relative amplitude error. The random errors are normally-distributed
% with zero mean and user-set standard deviation. Each subaperture
% contains numMX by numMY subarrays, each of which, like the
% subapertures, is time-steered and has a random absolute time error and
% a random relative amplitude error.  Finally, each subarray contains
% numNX by numNY elements, each of which is phase-steered and has a
% random absolute phase error and a random relative amplitude error.
% All elements are spaced by dx in x and dy in y, even across subarrays
% and subapertures.
%
% Set parameters of subapertures
%
numLX =  1;                % INPUT number of subapertures in x
numLY =  1;                %   and y dimensions
qntTimeL = 0;              % INPUT quantization interval in ns; 0 for no quantization
ofsTimeL = zeros (numLY, numLX);  % INPUT deterministic absolute time error in each subaperture in ns
stdTimeL = 0.000;          % INPUT standard deviation of absolute time error in each subaperture in ns
stdAmplL = 0.0;            % INPUT standard deviation of relative amplitude error
%
% Set parameters of subarrays
%
numMX =  8;                % INPUT number of subarrays in x
numMY =  8;                %   and y dimensions per subaperture
qntTimeM = 0;              % INPUT quantization interval in ns; 0 for no quantization
stdTimeM = 0.00;           % INPUT standard deviation of absolute time error in each subarray in ns
stdAmplM = 0.0;            % INPUT standard deviation of relative amplitude error
%
% Set parameters of elements
%
dx = 1.6 * 0.0254;         % INPUT element spacing in x
```

```
dy = 1.6 * 0.0254;       %   and y dimensions in meters
numNX =  8;              % INPUT number of elements in x
numNY =  8;              %   and y dimensions per subarray
qntPhseN =  0 * rpd;     % INPUT quantization interval in radians; 0 for no quantization
stdPhseN =  0 * rpd;     % INPUT standard deviation of absolute phase error in each element in radians
stdAmplN = 0.0;          % INPUT standard deviation of relative amplitude error
%
% Set other parameters
%
diamond = 0;             % INPUT 0: full array; 1: eliminate excitations on even diagonals
azmthOffst =   0 * rpd;  % INPUT angle of the positive x axis of the array above the azimuth reference

% Alternatively, select an array
%
switch 0  % INPUT case number for arrays below or 0 to use values above
  case 1
    % Insert frequency, array, and error parameters here
  case 2
end

% Specify steering angle
%
% Two coordinate systems, described below, are available for specifying
% the steering angle: traditional azimuth/elevation coordinates and
% spherical coordinates.  Azimuth/elevation coordinates are converted to
% spherical coordinates for internal program use, and output is given in
% spherical coordinates.
%
% In the diagrams below, the antenna lies in the x'-y' plane with
% boresight along the positive z' axis.  The antenna's z axis coincides
% with z', and its x axis is at an angle azmthOffst above the x' axis,
% which is the azimuth reference.
%
% The equations relating the the Cartesian coordinates x', y', and z' of
% a unit vector, the spherical coordinates Polar and Azmth, and the
% traditional coordinates Az and El are
%
%      x' = sin Polar cos Azmth = -cos El sin Az
%      y' = sin Polar sin Azmth =  sin El
%      z' = cos Polar           =  cos El cos Az
%
%      cos Polar =  cos El cos Az   = z'
%      tan Azmth = -tan El / sin Az = y' / x'
%
%      -tan Az = tan Polar cos Azmth = x' / z'
%       sin El = sin Polar sin Azmth = y'
%
% Since (x', y', z') is a unit vector, its components are direction
% cosines.
%
switch 2
  case 1
    %                     y'           Use azimuth/elevation
    %                    |             coordinates.  Given the
    %                 _--|             projection of the steering vector
    %                /   |             onto the x'-z' plane, the azimuth
    %       main __ /    |             is the angle between it and the
    %       beam   --__  |             z' axis, while the elevation is
    %             /    --o---------    the angle between it and the
    %          El |  _- /         x'   steering vector.  In the diagram,
    %             |_-  /               both angles are positive.
    %            _--__/
    % proj. in _-    /
    % xz plane     Az  z'
    %
    steerAz =  30 * rpd;               % INPUT azimuth angle
    steerEl =  30 * rpd;               % INPUT elevation angle
    steerPolar = acos  (cos (steerEl) * cos (steerAz));
    steerAzmth = atan2 (tan (steerEl), -sin (steerAz));
    %
  case 2
    %         proj. in    y'           Use spherical coordinates.  The
    %         xy plane   |             polar angle is the angle
    %                \_--|--_          between the z' axis and the
    %                 \  |   \ Azmth   steering vector.  The azimuth
    %       main __   /\ |    \        angle is the angle between the
    %       beam   --/_ \|    |        x' axis and the projection of
    %              | | --o--------     the steering vector onto the
    %              | |  /         x'   x'-y' plane.  In the diagram,
    %         Polar \ \/               both angles are positive.
    %                \/
    %                /
    %                 z'
    %
    steerPolar =   30 * rpd;           % INPUT polar angle
    steerAzmth =   60 * rpd;           % INPUT azimuth angle
    %
  otherwise
    error ('Invalid switch parameter.');
end
steerAzmth = steerAzmth - azmthOffst;  % now relative to antenna's x axis
```

```
% Set transform size
%
% (See comments elsewhere related to the discrete Fourier transform.)
%
tx = 2^9;  % INPUT number of transform points in x
ty = 2^9;  %   and y

% Declare independent variable and its values
%
% The main loop iterates over values of the independent variable named
% below.  The independent variable may be any parameter, including, for
% example, those describing geometry, frequency, error, quantization,
% and steering angle.  It may also be an otherwise unknown variable that
% is transformed to a known program parameter by custom code in the main
% loop.  To effectively disable the loop, set a dummy variable to a
% scalar value.
%
indVarName = 'stdTimeMPS';          % INPUT name of independent variable
indVar = 0 : 2 : 20;                % INPUT vector of values it will assume
indVar = indVar (:);                % make it a column vector
indVarLen = length (indVar);

% Initialize statistics variables
%
% If the excitations are random, one is often interested in the mean and
% standard deviation of the performance measures.  For each value of the
% independent variable, the program will generate numRlz realizations of
% the random excitations and accumulate the statistics of the
% performance measures.
%
numRlz =  16;                                     % INPUT number of realizations to generate
numAcc            = nan * ones (indVarLen, 1);   % number of realizations accumulated
pxS               = nan * ones (indVarLen, 2);
pyS               = nan * ones (indVarLen, 2);
pzS               = nan * ones (indVarLen, 2);
errPointS         = nan * ones (indVarLen, 2);
% errPointUncS      = nan * ones (indVarLen, 2);  % see later calculation of pointing error
beamPowerDBS      = nan * ones (indVarLen, 2);
beamDepthDBS      = nan * ones (indVarLen, 2);
hpbwMjrS          = nan * ones (indVarLen, 2);
hpbwMnrS          = nan * ones (indVarLen, 2);
rollS             = nan * ones (indVarLen, 2);
directivityDBS    = nan * ones (indVarLen, 2);
powerMainVisbDBS  = nan * ones (indVarLen, 2);
powerMainSideDBS  = nan * ones (indVarLen, 2);
powerVisbSideDBS  = nan * ones (indVarLen, 2);
powerSideAvgDBS   = nan * ones (indVarLen, 2);
slNrstDistS       = nan * ones (indVarLen, 2);
slNrstPowrDBS     = nan * ones (indVarLen, 2);
slLgstDistS       = nan * ones (indVarLen, 2);
slLgstPowrDBS     = nan * ones (indVarLen, 2);

% Initialize graphics
%
cmap = jet;         % INPUT color map
invertBkgd = 0;     % INPUT 0: figure background is lowest value of the colormap;
                    %       1: "                             highest
sumBW = 0;          % INPUT 0: summary plot in color; 1: in black and white
figPat = 1;         % INPUT figure number for power pattern
figSum = 2;         %   and summary
cbarVert = 0;       % INPUT 0 for horizontal bar below pattern, 1 for vertical to right
%
% Set window positions
%
cbarSize = 0.15;  % colorbar size relative to pattern
marginWid = 8;    % width margin in pixels
marginHgt = 44;   % height margin in pixels
screenSize = get (0, 'screenSize');
screenWid = screenSize (3);
screenHgt = screenSize (4);
figPatWid = 0.45 * (screenSize (3) - 4 * marginWid);       % width of pattern figure
figSumWid = screenSize (3) - 4 * marginWid - figPatWid;    % width of summary figure
if cbarVert
  figPatHgt = figPatWid;                                   % height of pattern figure
  figPatWid = figPatWid * (1 + cbarSize);
else
  figPatHgt = figPatWid * (1 + cbarSize);
end
figure (figPat);                      % create and position pattern figure
figPat = gcf;                         % update in case figPat couldn't be created
set (figPat, ...
  'position', [marginWid                       ...
               screenHgt-marginHgt-figPatHgt   ...
               figPatWid                       ...
               figPatHgt], ...
  'paperUnits', 'inches')
if cbarVert
  set (figPat, ...
    'PaperOrientation', 'landscape', ...
    'PaperPosition', [0.5 0.5 10 7.5]);
else
  set (figPat, ...
```

```
      'PaperOrientation', 'portrait', ...
      'PaperPosition', [0.5 0.5 7.5 10]);
  end
  clf;
  figure (figSum);                    % create and position summary figure
  figSum = gcf;                       % update in case figSum couldn't be created
  set (figSum, ...
    'position', [screenWid-marginWid-figSumWid ...
                 marginHgt                     ...
                 figSumWid                     ...
                 screenHgt-2*marginHgt]);
  orient tall;
  clf;
  %
  % Set pattern figure colors
  %
  if invertBkgd
    bkgd = cmap (size (cmap, 1), :);  % take background from high...
  else
    bkgd = cmap (1, :);               %   or low end of colormap
  end
  whitebg (figPat, bkgd);             % set pattern colors
  set (figPat, 'color', bkgd);        % override default background of whitebg
  %
  % Set summary figure colors and linestyles
  %
  whitebg (figSum, 'w');              % set summary colors
  set (figSum, 'color', 'w');         % override default background of whitebg
  if sumBW
    discrim = 'lineStyle';
    discrimValue = {'-'; '--'; ':'};  % line styles
    discrimName = discrimValue;       % names for legends will be same as style codes
    set (figSum, ...                  % black color forces cycling through line styles
      'DefaultAxesLineStyleOrder', discrimValue, ...
      'DefaultAxesColorOrder', [0 0 0]);
  else
    discrim = 'color';
    set (figSum, ...                                     % reset (if previously b&w, for
      'defaultAxesColorOrder', 'default', ...            %   example)
      'defaultAxesLineStyleOrder', 'default');
    discrimValue = get (figSum, 'defaultAxesColorOrder'); % put colors in array of RGB coordinates
    colorOrderHSV = rgb2hsv (discrimValue);              % equivalent HSV coordinates
    discrimValue = num2cell (discrimValue, 2);           % put each row in a cell
    colorNames = ['RYGCBMKW']';                          % first six by hue, then black & white
    discrimName = colorNames (1 + ...                    % convert H to an integer 0..5
      mod (round (6 * colorOrderHSV (:, 1)), 6));        %   then index into colorNames
    unsat = find (colorOrderHSV (:, 2) <= 0.25);         % unsaturated (gray) colors
    if ~isempty (unsat)
      discrimName (unsat) = colorNames (7 + ...          % threshold V to 0 (K) or 1 (W)
        (colorOrderHSV (unsat, 3) > 0.5));               %   then index into colorNames
    end
    discrimName = cellstr (discrimName);                 % put each letter in a cell
    clear colorOrderHSV colorNames unsat;
  end
  %
  % Set miscellaneous common properties
  %
  set ([figPat figSum], ...
    'invertHardCopy', 'off', ...
    'defaultTextFontSize', 8, ...
    'defaultAxesFontSize', 8, ...
    'toolbar', 'none');
  drawnow;
  clear marginWid marginHgt screenSize screenWid screenHgt;
  clear figPatWid figSumWid figPatHgt bkgd;

  % Loop over independent variable
  %
  for indx = 1 : indVarLen

    % Set value of independent variable
    %
    eval ([indVarName ' = indVar (indx);']);  % set variable to value
    %
    % Code may be inserted below to transform the independent variable to
    % known program variables.
    stdTimeM = stdTimeMPS * 1e-3;

    % Define labels for substructures
    %
    % Here we construct row and column vectors (corresponding to x and y,
    % respectively) that tell to which subaperture, subarray, or element a
    % given position corresponds.  The sample vectors are for numLX =
    % numLY = 2, numMX = numMY = 3, and numNX = numNY = 2.
    %
    numLMX = numLX * numMX;             % total number of subarrays
    numLMY = numLY * numMY;
    numLMNX = numLMX * numNX;           % total number of elements
    numLMNY = numLMY * numNY;
    nx =  0 : numLMNX - 1;              % e.g., [0 1 2 3 4 5 6 7 8 9 10 11]
    ny = (0 : numLMNY - 1)';            %       [0 1 2 3 4 5 6 7 8 9 10 11]'
    mx = floor (nx / numNX);            %       [0 0 1 1 2 2 3 3 4 4  5  5]
```

```
my = floor (ny / numNY);             %       [0 0 1 1 2 2 3 3 4 4  5  5]'
lx = floor (nx / (numMX * numNX));   %       [0 0 0 0 0 0 1 1 1 1  1  1]
ly = floor (ny / (numMY * numNY));   %       [0 0 0 0 0 0 1 1 1 1  1  1]'

% Calculate ideal (error-free) excitation magnitudes
%
excMagIdl = ones (numLMNY, numLMNX);               % uniform weighting
if diamond                                         % zero every other element
  excMagIdl = excMagIdl ...
    .* rem (ones (numLMNY, 1) * nx + ny * ones (1, numLMNX), 2);
end
excMagIdl = excMagIdl / sum (excMagIdl(:).^2);  % normalize to unit power

% Calculate ideal (error-free) excitation phases
%
% The ideal excitation phases are those that produce perfectly
% constructive interference in the direction of the steering vector at
% the reference frequency.  This implies a linear phase progression
%
%      excPhsIdl = -k0 (dx sx nx + dy sy ny) + const.,
%
% where k0 (= 2 pi fRef) is the reference wave vector.  Quantization
% prevents the array from achieving this flat phase front for all
% steering angles.  However, the beamformer simulated here mitigates
% effects due to subaperture quantization by adjusting the subarray
% delays, which is effective if the subarrays are quantized at a finer
% interval than the subapertures.  Likewise, it compensates for
% subarray quantization with the element phasers.
%
% We choose the constant in the phase progression to be zero, which
% means that the lowest and leftmost components (those for which lx,
% ly, mx, my, nx, or ny is zero) are never delayed, while the highest
% and rightmost components have delays that depend strongly on the
% steering vector.
%
sx = sin (steerPolar) * cos (steerAzmth);         % direction cosines
sy = sin (steerPolar) * sin (steerAzmth);         %   for steering
sz = cos (steerPolar);                            % used much later
TimeLX = -dx * sx * numNX * numMX * lx / c;
TimeLY = -dy * sy * numNY * numMY * ly / c;
TimeL = ones (numLMNY, 1) * TimeLX + TimeLY * ones (1, numLMNX);
if qntTimeL ~= 0                                  % quantize?
  TimeL = round (TimeL / qntTimeL) * qntTimeL;    % yes
end;
TimeMX = -dx * sx * numNX * mx / c;
TimeMY = -dy * sy * numNY * my / c;
TimeM = ones (numLMNY, 1) * TimeMX + TimeMY * ones (1, numLMNX) - TimeL;
if qntTimeM ~= 0
  TimeM = round (TimeM / qntTimeM) * qntTimeM;
end;
TimeNX = -dx * sx * nx / c;
TimeNY = -dy * sy * ny / c;
PhseN = twopi * fRef * ...
  (ones (numLMNY, 1) * TimeNX + TimeNY * ones (1, numLMNX) - TimeL - TimeM);
if qntPhseN ~= 0
  PhseN = round (PhseN / qntPhseN) * qntPhseN;
end;
excPhsIdl = twopi * fOpr * (TimeL + TimeM) + PhseN;
clear TimeLX TimeLY TimeL TimeMX TimeMY TimeM TimeNX TimeNY TimeN PhseN;

% Prepare transform mapping
%
% The far-field array factor is the Fourier transform of the complex
% excitations.  Considering the x dimension only (the operation in the
% y dimension is analogous), the discrete Fourier transform (DFT) used
% later calculates the far-field array factor at the direction cosine
% cx as
%
%                 tx-1
%      g (cx ) = sum  exp (-i k cx dx q) e ,
%                 q=0                     q
%
% where the e[q] are the complex excitations (zero-padded if tx
% exceeds the array size), k (= 2 pi / lambda) is the operating wave
% vector, and lambda (= c / fOpr) is the operating wavelength.  The
% argument of the exponential in the transform may be written -i 2 pi
% (cx / lambda) * (dx q), where cx / lambda is the spatial frequency
% and dx q is the spatial coordinate.  The direction cosines for which
% g is calculated in the DFT are
%
%              lambda p
%      cx  = -------- , p = 0, 1, ..., tx - 1
%        p     dx tx
%
% so that the argument of the exponential is -i 2 pi p q / tx.  We
% have
%
%      g (cx     ) = g (cx )  for any integer p.
%           p+tx          p
%
% Given g (cx[p]), the array factor may be obtained at any angle using
%
```

```
%                tx-1   / p    dx cx        \
%     g (cx) = sum  S | -- - ------ , tx | g (cx )
%                p=0    \ tx   lambda       /      p
%
% where the geometric progression
%
%                  1  tx-1
%     S (x, tx) = -- sum  exp (i 2 pi q x)
%                 tx q=0
%
%                  1  exp (i 2 pi x tx) - 1
%               = -- -----------------------
%                 tx   exp (i 2 pi x) - 1
%
% is an interpolating function, but this formula is not used below.
% Incidentally, note that
%
%                    1  | sin (tx pi x) |
%     |S (x, tx)| = -- | ------------- | ,
%                   tx |  sin (pi x)   |
%
% an expression that often appears in array theory.
%
% After the DFT is calculated, the program maps the results into the
% region of cx-cy space where the direction cosines have magnitude 1
% or less, tiling as necessary to fill the region.  The portion of
% that region for which cx^2 + cy^2 <= 1 corresponds to visible real
% space (radiating waves).  Later processing requires a border of at
% least one element outside the visible region. This section prepares
% the mapping.
%
txLim = tx * dx * fOpr / c;                 % values of p (not necessarily
tyLim = ty * dy * fOpr / c;                 %   integer) for which cx and cy are 1
txIndxLimMin = -floor (txLim      ) - 1;    % largest integers p for which
tyIndxLimMin = -floor (tyLim      ) - 1;    %   cx, cy > -1
txIndxLimMax =  floor (txLim + 0.5) + 1;    % smallest integers p for which cx,
tyIndxLimMax =  floor (tyLim + 0.5) + 1;    %   cy > (1 + half element spacing)
  % Extra half element spacing is needed only for surface plots with
  % flat shading; see graphics code below
ax = txIndxLimMax - txIndxLimMin + 1;       % number of angle samples in x
ay = tyIndxLimMax - tyIndxLimMin + 1;       %   and y
txIndx =  txIndxLimMin : txIndxLimMax  ;    % row vector of indices
tyIndx = (tyIndxLimMin : tyIndxLimMax)';    % column vector
dirCosX = txIndx / txLim;                   % corresponding direction cosines
dirCosY = tyIndx / tyLim;                   %   cx and cy
dirCosShiftX = (txIndx - 0.5) / txLim;      % shifted direction cosines for use
dirCosShiftY = (tyIndx - 0.5) / tyLim;      %   with flat shading
txIndx = txIndx - tx * floor (txIndx / tx); % zero-based indices into columns
tyIndx = tyIndx - ty * floor (tyIndx / ty); %   (x) and rows (y) of DFT results
  % The above lines accomplish the tiling function by folding the
  % indices into the interval [0, tx - 1]
tIndx = tyIndx * ones (1, ax) ...           % indices to elements (one-based,
  + ones (ay, 1) * txIndx * ty + 1;         %   column-ordered)
clear txIndxLimMin tyIndxLimMin txIndxLimMax tyIndxLimMax txIndx tyIndx;

% Prepare far-field angle mapping
%
% Physically, the array factor is a function of position on a
% hemisphere.  The direction cosines used in the Fourier transform are
% the x and y coordinates of points on the unit hemisphere.  Below we
% determine the region of the transform results that corresponds to
% visible space, namely cx^2 + cy^2 <= 1, and calculate the z
% coordinates of points in visible space according to
%
%                   2    2 1/2
%     cz = (1 - cx  - cy )    , Re cz >= 0.
%
% For points outside visible space, cz is set to zero.
%
dirCosXMtx = ones (ay, 1) * dirCosX;                 % now a matrix
dirCosYMtx = dirCosY * ones (1, ax);
radSqr = dirCosXMtx.^2 + dirCosYMtx.^2;              % squared radius
visBool = logical (radSqr < 1);                      % 1 in visible space, 0 elsewhere
dirCosZMtx = zeros (ay, ax);                         % 0 outside visible space
dirCosZMtx (visBool) = sqrt (1 - radSqr (visBool));  % positive in visible space
clear radSqr;

% Loop over realizations
%
for rlzNum = 1 : numRlz

  % Calculate excitation magnitudes with error
  %
  excMagErr = 1 + stdAmplN * randn (numLMNY, numLMNX);  % element-level error
  err       = 1 + stdAmplM * randn (numLMY , numLMX );  % temporary matrix
  excMagErr = excMagErr .* err (my + 1, mx + 1);        % subarray-level error
  err       = 1 + stdAmplL * randn (numLY  , numLX  );  % temporary
  excMagErr = excMagErr .* err (ly + 1, lx + 1);        % subaperture-level error
  excMag    = excMagErr .* excMagIdl;                   % actual (with error) magnitude
  clear err excMagErr;

  % Calculate excitation phases with error
```

```
%
excPhsErr = stdPhseN * randn (numLMNY, numLMNX);      % element-level error
err       = stdTimeM * randn (numLMY , numLMX ) ...   % temporary matrix
  * twopi * fOpr;                                     %   (equivalent phase)
excPhsErr = excPhsErr + err (my + 1, mx + 1);         % subarray-level error
err       = (ofsTimeL + stdTimeL * randn (numLY, numLX)) * twopi * fOpr;
excPhsErr = excPhsErr + err (ly + 1, lx + 1);         % subaperture-level error
excPhs    = excPhsErr + excPhsIdl;                    % actual (with error) phases
clear err excPhsErr;

% Assemble complex excitations
%
exc = excMag .* exp (-i * excPhs);
% clear excMag excPhs;

% Calculate the field pattern (array factor)
%
% Here the DFT is calculated and the result rearranged into the
% desired region of direction cosine space.  The element factor is
% unity.  Mutual coupling is ignored.
%
g = fft2 (exc, ty, tx);          % first element is zero frequency
g = g (tIndx);                   % rearrange
gSqr = real (conj (g) .* g);     % squared magnitude
gMag = sqrt (gSqr);              % magnitude
clear g;

% Determine actual beam direction by fitting the transform
%
% This first of two methods for calculating the beam pointing vector
% uses the information in the Fourier transform of the excitations.
% First we locate the element of g with the largest value.  (If the
% maximum value of elements is obtained by more than one element,
% this code will use the one with the smallest column index and the
% smallest row index within that column.  Later processing will
% determine whether the multiple maxima are all within the main
% beam.)  To estimate the location of the maximum of the underlying
% continuous function, that element and its eight neighbors are
% fitted to the elliptic paraboloid [1]
%
%      1   2              1   2
%      - U x  + W x y + - V y  + X x + Y y + Z = |g (x, y)|
%      2                  2
%
% (in the direction cosine coordinate system) in a least-squares
% sense.  We employ the technique of QR decomposition to find the
% least-squares solution.  Define the solution vector
%
%                          T
%      a = [U W V X Y Z]
%
% of length M = 6, the ordinate (column) vector b with elements
%
%      b  = |g (x , y )|
%       i         i   i
%
% of length N = 9 (number of fitted points), and the design matrix A
% having rows
%
%             1  2        1  2
%      A   = [ - x   x y   - y   x   y   1 ] ,
%       i,:    2 i   i i  2 i   i   i
%
% where the (x[i], y[i]) are the coordinates of the points
% neighboring and including the maximum element.  QR decomposition
% of A factorizes it as
%
%      A = Q R ,
%
% where Q is unitary (Q' Q = eye) and R is upper triangular.  (We
% use Matlab's economy size decomposition, for which Q is M-by-N and
% R is N-by-N.)  The least-squares solution of
%
%      A a = b
%
% is given by
%
%            -1
%      a = R   Q' b .
%
% The error in the solution depends on the degree to which the
% values of |g| depart from parabolic form, which is indicated by
% the reduced chi-square
%
%         2   1   2
%      chi   = -- chi ,
%         nu  nu
%
% where nu = N - M is the number of degrees of freedom and
%
```

```
%          2
%      chi = (A a - b)' (A a - b) .
%
% The covariance matrix for the solution vector is normally given by
% the matrix inverse of the curvature matrix
%
%      alpha = A' A ,
%
% whose elements are the second partial derivatives of chi^2 with
% respect to the elements of the solution vector [2]:
%
%               1    d       d        2
%      alpha    = - ------ ------ chi   .
%           mn   2 d a[m] d a[n]
%
% To incorporate the degree of deviation from parabolic form, we
% scale the covariance matrix by the reduced chi-square, as
%
%                  2         -1       2        -1
%      C      = chi    (A' A)    = chi    (R' R)   .
%       poly       nu                nu
%
% This completes the least-squares procedure.
%
% Having obtained the coefficients of the best-fit polynomial, we
% locate its maximum.  The coordinates (plx, ply) of the maximum
% solve
%
%      / U W \ / plx \    / X \    / 0 \
%      |     | |     | +  |   |  = |   | ;
%      \ W V / \ ply /    \ Y /    \ 0 /
%
% that is,
%
%      / plx \    1 /  V -W \ / X \
%      |     | = - |        | |   | ,
%      \ ply /    D \ -W  U / \ Y /
%
% where
%
%           2
%      D = W  - U V
%
% is the discriminant of the polynomial (and the negative
% determinant of the matrix).  If the maximum so found lies outside
% the interpolation region, the region is expanded by one sample in
% each direction and the least-squares fit is repeated.  This loop
% continues until the interpolation region exceeds a certain size or
% a satisfactory maximum is found.  Assuming a maximum has been
% found, the covariance matrix for plx and ply is calculated next.
% We first form the derivative matrix or Jacobian
%
%           d (plx, ply)
%      J  = ------------
%       p       d (a)
%
%          / d plx d plx d plx d plx d plx d plx \
%          | ----- ----- ----- ----- ----- ----- |
%          |  d U   d W   d V   d X   d Y   d Z  |
%        = |                                     | .
%          | d ply d ply d ply d ply d ply d ply |
%          | ----- ----- ----- ----- ----- ----- |
%          \  d U   d W   d V   d X   d Y   d Z  /
%
% The plx derivatives are
%
%      d plx    V X - W Y
%      ----- = --------- V
%       d U       D^2
%
%      d plx    W^2 Y + U V Y - 2 W V Y
%      ----- = -----------------------
%       d W              D^2
%
%      d plx    W X - U Y
%      ----- = --------- W
%       d V       D^2
%
%      d plx    V
%      ----- = -
%       d X     D
%
%      d plx     W
%      ----- = - - ,
%       d Y      D
%
% and the ply derivatives follow by exchanging U for V and X for Y
% everywhere.  The covariance matrix for plx and ply is simply
%
%                      T
%      C  = J  C     J  .
%       p    p  poly  p
```

```
%
% If no satisfactory maximum was found by the above procedure, the
% location of the maximum of |g| is used.  A covariance matrix is
% fabricated for which the area of the 2 sigma ellipse equals the
% area of four grid squares to indicate the uncertainty in the
% actual location of the maximum.
%
% [1] D. H. von Seggern, _CRC Standard Curves and Surfaces_.  Boca
% Raton, FL: CRC, 1993.
%
% [2] P. R. Bevington and D. K. Robinson, _Data Reduction and Error
% Analysis for the Physical Sciences_, 2nd ed.  New York, NY:
% McGraw-Hill, 1992, pp. 121-125.
%
[gSqrMaxAct, gSqrMaxRow] = max (gSqr .* visBool);     % maximum values and their rows
[gSqrMaxAct, gSqrMaxCol] = max (gSqrMaxAct);          % overall maximum and column
gSqrMaxRow = gSqrMaxRow (gSqrMaxCol);                 % corresponding row
intRad = 1;                                           % interpolation radius
p1OK = 0;
while ~p1OK & intRad < 4
  p1Nbrs = [-intRad : intRad];
  p1x = dirCosXMtx (gSqrMaxRow + p1Nbrs, gSqrMaxCol + p1Nbrs);  % neighbors
  p1y = dirCosYMtx (gSqrMaxRow + p1Nbrs, gSqrMaxCol + p1Nbrs);
  p1z = gMag       (gSqrMaxRow + p1Nbrs, gSqrMaxCol + p1Nbrs);
  p1x = p1x (:);
  p1y = p1y (:);
  p1z = p1z (:);
  pDesMtx = [p1x.*p1x/2 p1x.*p1y p1y.*p1y/2 ...       % design matrix
    p1x p1y ones(size(p1x))];
  [Q, R] = qr (pDesMtx, 0);                           % now pDesMtx = Q * R; Q' * Q = eye
  pPoly = R \ (Q' * p1z);                             % solves pDesMtx * pPoly = p1z
  dof = length (p1z) - length (pPoly);                % degrees of freedom in the fit
  chiSqr = p1z - pDesMtx * pPoly;                     % deviations only
  chiSqr = chiSqr' * chiSqr;                          % now sum of squared deviations
  pPolyVar = inv (R' * R) * chiSqr / dof;             % pDesMtx' * pDesMtx = R' * R
  pVec = -[pPoly(1) pPoly(2); pPoly(2) pPoly(3)] ...  % find critical point
    \ [pPoly(4); pPoly(5)];
  p1OK = ...                                          % is interpolated point
      (pVec (1) > dirCosX (gSqrMaxCol - intRad)) ...  %   inside neighborhood?
    & (pVec (1) < dirCosX (gSqrMaxCol + intRad)) ...  %   (pathological cases can
    & (pVec (2) > dirCosY (gSqrMaxRow - intRad)) ...  %   place it outside)
    & (pVec (2) < dirCosY (gSqrMaxRow + intRad));
  if ~p1OK
    intRad = intRad + 1;
  end
end
if p1OK
  p1x = pVec (1);                                     % keep interpolated point
  p1y = pVec (2);
  pDet = pPoly (2)^2 - pPoly (1) * pPoly (3);         % determinant
  p1xDer = [[1 -1]*(pPoly([3 2]).*pPoly([4 5]))*pPoly(3)/pDet
            [1 1 -2]*(pPoly([2 1 2]).*pPoly([2 3 3]).*pPoly([5 5 4]))/pDet
            [1 -1]*(pPoly([2 1]).*pPoly([4 5]))*pPoly(2)/pDet
            pPoly(3)
            -pPoly(2)
            0 ]' / pDet;                              % derivatives
  p1yDer = [[1 -1]*(pPoly([2 3]).*pPoly([5 4]))*pPoly(2)/pDet
            [1 1 -2]*(pPoly([2 3 2]).*pPoly([2 1 1]).*pPoly([4 4 5]))/pDet
            [1 -1]*(pPoly([1 2]).*pPoly([5 4]))*pPoly(1)/pDet
            -pPoly(2)
            pPoly(1)
            0 ]' / pDet;
  p1Var = [p1xDer; p1yDer] * pPolyVar * [p1xDer; p1yDer]';  % covariance matrix
else                                                  % interpolated point is outside
  p1x = dirCosXMtx (gSqrMaxRow, gSqrMaxCol);          % use location of
  p1y = dirCosYMtx (gSqrMaxRow, gSqrMaxCol);          %   actual maximum
  p1Var = diag (1 ./ (pi * [txLim tyLim].^2));        % 2 sigma area = 4 grid squares
end
clear intRad p1OK p1Nbrs p1z pDesMtx Q R pPoly;
clear dof chiSqr pPolyVar pVec pDet p1xDer p1yDer;

% Determine actual beam direction by fitting the excitation phases
%
% The second method for calculating the pointing vector uses the
% excitation magnitudes and phases, not the transform.  For brevity,
% define
%
%        Delta  (x, y) = k n dx x + k m dy y + theta   ,
%             mn                                   mn
%
% where the theta[m,n] are the excitation phases, and let e[m,n]
% denote the excitation magnitudes.  When we express the power
% pattern as
%
%                   2
%      |g (x, y)|    = sum sum sum sum e       e
%                      m1  n1  m2  n2  m1,n1  m2,n2
%
%                    * exp [i (Delta      - Delta      )]
%                                  m1,n1          m2,n2
%
```

```
%                              2
%                     = sum  e     + 2  sum   sum' e       e
%                       m,n   m,n      m1,n1 m2,n2  m1,n1  m2,n2
%
%                       * cos (Delta       - Delta      ) ,
%                                   m1,n1          m2,n2
%
% where the primed sum is over distinct pairs (m1, n1) and (m2, n2),
% we see that the maximum occurs where the cosine contributions are
% largest.  If the phase front is nearly flat, the arguments of the
% cosines will be small for (x, y) near the direction of phase front
% propagation.  To fourth order in the arguments,
%
%                  2              2
%      |g (x, y)|  = (sum e  )  + 2  sum   sum  e       e
%                     m,n  mn       m1,n1 m2,n2  m1,n1  m2,n2
%
%                        /  1                            2
%                      * |  - - (Delta        - Delta      )
%                        \  2         m1,n1          m2,n2
%
%                           1                            4 \
%                         + -- (Delta        - Delta      )  | .
%                           24        m1,n1          m2,n2   /
%
% Keeping terms to only second order, we are motivated to find the x
% and y (implicit in Delta[mn]) that minimize
%
%          2                                                       2
%      chi  =  sum    sum  e       e        (Delta        - Delta      ) ,
%          2   m1,n1 m2,n2  m1,n1  m2,n2          m1,n1          m2,n2
%
% where the subscript 2 denotes the second-order truncation.  Taking
% derivatives with respect to x and y and rearranging yields the
% normal equations
%
%      / 0 \                                 / n1-n2 \
%      |   | =  sum    sum  e       e        |       |
%      \ 0 /   m1,n1 m2,n2  m1,n1  m2,n2 \ m1-m2 /
%
%                  /                                             \
%                * | [n1-n2 m1-m2] Pi + theta        - theta      | ,
%                  \                         m1,n1          m2,n2 /
%
% where Pi = k [dx*x dy*y]'.  Here the sums contain a total of
% (numLMNX^2 numLMNY^2) terms, which may be of the order of one
% million for a typical array.  To reduce this number, we transform
% the least-squares problem to an equivalent but simpler problem.
% First, the above normal equations may be rewritten with the column
% vector [n1-n2 m1-m2]' replaced by [n1 m1]', which may be seen by
% separating the column vector into two sums and exchanging (m1,n1)
% with (m2,n2) in one of the sums.  Second, the row vector may be
% separated into two sums to obtain
%
%      / 0 \                      / n \
%      |   | = ( sum e  ) sum e    |   | ( [n m] Pi + theta  )
%      \ 0 /     m,n  mn  m,n  m,n \ m /                   mn
%
%                 /          / n \ \
%               - | sum e    |   | | sum e    ( [n m] Pi + theta  ) .
%                 \ m,n  mn \ m / / m,n  mn                     mn
%
% Based on the second term, we define
%
%                 sum e    ( [n m] Pi + theta   )
%                 m,n  mn                     mn
%      Delta = - ------------------------------ .
%                              sum e
%                              m,n  mn
%
% Finally, we may rewrite the normal equations as
%
%      / 0 \             / n \
%      | 0 | = sum e     | m | ( [n m 1] Gamma + theta  ) ,
%      \ 0 /   m,n  m,n  \ 1 /                        mn
%
% where Gamma = [k*dx*x k*dy*y Delta]' and the third row follows
% from the definition of Delta.  These normal equations contain only
% (numLMNX numLMNY) terms, nominally on the order of 1000.  They
% find the plane  k n dx x + k m dy y + Delta  that best fits
% -theta[m,n] in a weighted least-squares sense.  The solution is
% found using QR decomposition of the design matrix, which has rows
% [n m 1].
%
% The error in the solution is determined not by the deviations of
% the -theta[mn] from the best-fit plane nor by the deviations
% Delta[m1,n1] - Delta[m2,n2] appearing in chi^2 earlier, for the
% solution is exactly the power pattern maximum to second order in
% the cosine arguments.  However, the error does depend on the
% fourth and higher powers of the cosine arguments.  So motivated,
% we consider the fourth-order merit function
%
```

```
%          2                                                       2
%     chi  =  sum   sum   e      e       (Delta        - Delta      )
%        4    m1,n1 m2,n2  m1,n1  m2,n2         m1,n1           m2,n2
%
%                 /      1                           2 \
%               * |  1 - -- (Delta        - Delta      )  |
%                 \      12        m1,n1         m2,n2   /
%
% and observe that chi4^2 =< chi2^2 everywhere.  We interpret the
% difference chi2^2 - chi4^2 as indicative of the error in the
% solution, and we scale the covariance matrix by that amount.  The
% covariance matrix used is the inverse of the curvature matrix for
% chi2^2; that curvature matrix is
%
%                  2
%       alpha = k  sum   sum   e      e
%                  m1,n1 m2,n2  m1,n1  m2,n2
%
%                  / (n1-n2)dx \
%                * |           | [ (n1-n2)dx (m1-m2)dy ] .
%                  \ (m1-m2)dy /
%
% Because we construct the design matrix for the simpler
% least-squares problem, we must construct alpha explicitly.
% However, this can be accomplished by analytically expanding the
% differences and factoring the sums.
%
phsX = ones (numLMNY, 1) * nx * dx * (twopi * fOpr / c);
phsY = ny * ones (1, numLMNX) * dy * (twopi * fOpr / c);
phsX = phsX (:);
phsY = phsY (:);
excMagV = excMag (:);
desMtx = [phsX phsY ones(numLMNX*numLMNY,1)] ...
  .* (sqrt (excMagV) * ones (1, 3));
[Q, R] = qr (desMtx, 0);  % now desMtx = Q * R and Q' * Q = eye [conj. transpose]
excPhsWgt = -excPhs (:) .* sqrt (excMagV);
p2Vec = R \ (Q' * excPhsWgt);  % desMtx * p2Vec = excPhsWgt
p2x = p2Vec (1);
p2y = p2Vec (2);
DeltaPhs = [phsX phsY ones(numLMNX*numLMNY,1)] * p2Vec + excPhs (:);
sumExcMagDelta1 = excMagV .* DeltaPhs;
sumExcMagDelta2 = sumExcMagDelta1 .* DeltaPhs;
sumExcMagDelta3 = sumExcMagDelta2 .* DeltaPhs;
sumExcMagDelta4 = sumExcMagDelta3 .* DeltaPhs;
sumExcMagDelta0 = sum (excMagV);
sumExcMagDelta1 = sum (sumExcMagDelta1);
sumExcMagDelta2 = sum (sumExcMagDelta2);
sumExcMagDelta3 = sum (sumExcMagDelta3);
sumExcMagDelta4 = sum (sumExcMagDelta4);
chiSqrRed = (  2 * sumExcMagDelta4 * sumExcMagDelta0 ...
             - 8 * sumExcMagDelta3 * sumExcMagDelta1 ...
             + 6 * sumExcMagDelta2 * sumExcMagDelta2 ) / 12;
chiSqrRed = max (0, chiSqrRed);  % in case of roundoff error
excMagPhs = excMagV' * [phsX phsY];
crvMtx = 2 * sumExcMagDelta0 * [phsX phsY]' ...
    * ([phsX phsY] .* [excMagV excMagV]) ...
  - 2 * excMagPhs' * excMagPhs;
p2Var = chiSqrRed * inv (crvMtx);
clear phsX phsY excMagV desMtx Q R excPhsWgt p2Vec DeltaPhs;
clear sumExcMagDelta0 sumExcMagDelta1 sumExcMagDelta2 sumExcMagDelta3 sumExcMagDelta4;
clear chiSqrRed excMagPhs crvMtx;

% Construct pointing vector
%
% Above we constructed two pointing vectors by different methods.
% The method of fitting the transform is robust even for large
% errors but limited by the transform resolution.  On the other
% hand, the method of fitting the excitation phases is independent
% of transform resolution but accurate only for small errors,
% approaching the exact solution as the phase errors decrease.  We
% wish to obtain a single pointing vector for subsequent use, and
% for this purpose we form a weighted average.  Specifically, each
% vector is weighted by the inverse of the area of its covariance
% ellipse, which is pi times the determinant of the covariance
% matrix.  Similarly, a single covariance matrix is obtained by
% weighing each covariance matrix by the square of the pointing
% vector weights, normalized to avoid effectively halving the
% covariance matrix when the two incoming matrices are nearly equal.
%
area1 = det (p1Var);
area2 = det (p2Var);
areaTot = area1 + area2;
wght1 = area2 / areaTot;
wght2 = area1 / areaTot;
px = wght1 * p1x + wght2 * p2x;
py = wght1 * p1y + wght2 * p2y;
pVar = (wght1^2 * p1Var + wght2^2 * p2Var) / (wght2^2 + wght1^2);
clear area1 area2 areaTot wght1 wght2;
clear p1x p1y p1Var;
clear p2x p2y p2Var;

% Calculate peak power density and pointing error
```

```
%
% To avoid inaccuracies due to interpolation, the peak power
% density is obtained by explicitly evaluating the Fourier
% transform at the pointing vector.  The pointing error is
% straightforwardly calculated from the dot product of the steering
% and pointing vectors.  However, an optional second method is
% coded that makes use of the pointing vector covariance matrix to
% calculate the uncertainty (standard deviation) of the pointing
% error due to uncertainty in the pointing vector.  If this
% uncertainty is desired, also uncomment lines elsewhere that refer
% to errPointUnc and errPointUncS.
%
pxy2 = px^2 + py^2;
peakVisb = (pxy2 <= 1);
if peakVisb
  gSqrMax = exp (-i * twopi * (fOpr / c) ...        % complex field
    * (  px * dx * ones (numLMNY, 1) * nx ...
       + py * dy * ny * ones (1, numLMNX))) .* exc;
  gSqrMax = sum (gSqrMax (:));                      % phaser sum
  gSqrMax = real (conj (gSqrMax) * gSqrMax);        % power
  pz = sqrt (1 - pxy2);
  if 0                                              % INPUT 0 to skip std dev, 1 to calc
    pCrs = [ 0  -sz  sy                             % cross product
             sz  0  -sx
            -sy  sx  0 ] * [px py pz]';
    pCrsDer = [  -sy*px/pz -sz-sy*py/pz             % rows correspond to
               sz+sx*px/pz     sx*py/pz             %   components of pCrs;
                -sy           sx         ];         %   columns, to pVec
    pCrsVar = pCrsDer * pVar * pCrsDer';            % covariance matrix
    pCrsMag = sqrt (pCrs' * pCrs);                  % magnitude
    if pCrsMag > 0
      pCrsMagDer = pCrs' / pCrsMag;                 % derivative exists
      pCrsMagVar = pCrsMagDer * pCrsVar * pCrsMagDer';
    else                                            % derivative doesn't exist
      pCrsMagVar = trace (pCrsVar) / 3;             % average of principal variances
    end
    errPoint = asin (pCrsMag);
    errPointUnc = abs (1 / sqrt (1 - pCrsMag^2)) * sqrt (pCrsMagVar);
    clear pCrs pCrsDer pCrsVar pCrsMag pCrsMagDer pCrsMagVar;
  else
    errPoint = acos (min (1, ...                    % dot product for error; min
      sx * px + sy * py + sz * pz));                %   prevents roundoff problems
    errPointUnc = nan;
  end
else                                                % maximum is invisible
  gSqrMax = gSqrMaxAct;
  px = nan;
  py = nan;
  pz = nan;
  errPoint = nan;
  errPointUnc = nan;
end
beamPowerDB = 10 * log10 (gSqrMax);
clear pxy2;

% Determine main beam region
%
% The angular domain of the main beam is constructed starting with
% the maximum element.  The largest neighboring element is added on,
% followed by the largest neighbor of either point, and so on.  This
% accretion continues until any neighbor of the largest element on
% the main beam border exceeds the element added previously.
% Effectively, elements are added with values descending from the
% peak until an opportunity to ascend is reached. All visible
% elements outside of the main beam are declared to be in the
% sidelobes.
%
beamWidLvl = gSqrMax / 2;                         % power level where beam width is measured
adjc = [-ay-1 -ay -ay+1 -1 1 ay-1 ay ay+1];       % relative indices of neighbors
  % Note: We must include the diagonal neighbors in order to
  % correctly descend a structure such as [1 0.4; 0.5 0.9].
beamBool = logical (zeros (ay, ax));              % build main beam in Boolean variable
brdrLen = 1;
brdrIndx = (gSqrMaxCol - 1) * ay + gSqrMaxRow;    % start with maximum
brdrVal = 0;                                      % any value will do here
beamBool (brdrIndx) = 1;                          % main beam begins with maximum
adjcIndx = brdrIndx + adjc';                      %   and neighbors
adjcIndx = adjcIndx (visBool (adjcIndx));         %   that are visible
adjcVal = gSqr (adjcIndx);                        % get values
beamDepth = inf;                                  % get the loop started
beamVisb = 1;
capVisb = (gSqrMaxAct > beamWidLvl);              % usually true unless resolution is too low
while max (adjcVal) <= beamDepth                  % are the new neighbors all downhill?
  brdrLen = brdrLen - 1;                          % yes; remove element from border
  brdrIndx = brdrIndx (1 : brdrLen);
  brdrVal  = brdrVal  (1 : brdrLen);
  beamBool (adjcIndx) = ones (size (adjcIndx));   % add neighbors to main beam
  for adjcPtr = 1 : length (adjcIndx)             %   and to border
    pos = sum (brdrVal <= adjcVal (adjcPtr));     % ordered least to greatest
    brdrIndx = [brdrIndx(1:pos) adjcIndx(adjcPtr) brdrIndx(pos+1:brdrLen)];
    brdrVal  = [brdrVal(1:pos)  adjcVal(adjcPtr)  brdrVal(pos+1:brdrLen) ];
    brdrLen = brdrLen + 1;
```

```
      end
      beamDepth = brdrVal (brdrLen);                    % pick largest element from border
      adjcIndx = brdrIndx (brdrLen) + adjc;             % neighbors of chosen element
      adjcIndx = adjcIndx (visBool (adjcIndx));         % eliminate invisible points
      if length (adjcIndx) < length (adjc)              % were some invisible?
        beamVisb = 0;                                   % yes; clear flag
        if beamDepth >= beamWidLvl                      % are we below the threshold?
          capVisb = 0;                                  % no; the cap is partially invisible
        end
      end
      adjcIndx = adjcIndx (~beamBool (adjcIndx));       % use only new elements
      adjcVal = gSqr (adjcIndx);                        % and get their values
    end
    capClosed = capVisb & (beamDepth < beamWidLvl);     % closed contour at beamWidLvl?
    sideBool = visBool & ~beamBool;                     % sidelobe region
    beamIndx = find (beamBool);
    if max (max (gSqr (sideBool))) < gSqrMaxAct         % duplicate maximum outside beam?
      beamExist = 1;                                    % no; the beam is identified
      if ~peakVisb
        disp ('Warning: The main beam peak is invisible; some calculations');
        disp ('  may return NaN.');
      elseif ~capVisb
        disp ('Warning: The beam width contour of the main beam is partially');
        disp ('  invisible; some calculations may return NaN.');
      elseif ~capClosed
        disp ('Warning: The main beam is insufficiently deep for obtaining');
        disp ('  its width; some calculations may return NaN.');
      elseif ~beamVisb
        disp ('Warning: The main beam is partially invisible; some');
        disp ('  calculations may return NaN.');
      end
    else
      beamExist = 0;                                    % yes; the beam is ambiguous
      disp ('Warning: The main beam is not identifiable; some ');
      disp ('  calculations will return NaN.');
      px = nan;                                         % strike earlier results
      py = nan;
      pz = nan;
      errPoint = nan;
      beamVisb = 0;
      beamIndx = [];
      peakVisb = 0;
      capVisb = 0;
      capClosed = 0;
      beamDepth = nan;
      sideBool = visBool;                               % treat visible space as sidelobes
    end
    if beamDepth == 0
      beamDepthDB = -inf;                               % avoid warning message
    else
      beamDepthDB = 10 * log10 (beamDepth / gSqrMax);
    end
    clear adjc beamBool brdrLen brdrIndx brdrVal adjcIndx adjcVal beamDepth adjcPtr pos;

    % Determine main beam width and roll
    %
    % The analysis of the beam's width and roll is conducted using a
    % stereographic projection, for which projections of great circles
    % intersect at the same angles as the great circles on a sphere.
    % (See the comments in the plotting section below for details.)
    % This property allows us to obtain, in the limit of a narrow beam,
    % the correct roll angle and the beam widths along two orthogonal
    % great circles.
    %
    % The actual calculations are based on fitting the half-power contour
    % of the main beam to an ellipse.  First the contour is obtained in
    % direction cosine space, then the coordinates are transformed to
    % stereographic coordinates.  The contour is fitted to the conic
    % section
    %
    %      1   2            1   2
    %      - U x  + W x y + - V y  + X x + Y y + Z = 0
    %      2                2
    %
    % using a simple algorithm that minimizes the algebraic distance as
    % follows.  Define the design matrix D to have rows
    %
    %                1  2            1  2
    %      D    = [ ( - x  ) (x y ) ( - y  )  x    y   1] ,
    %       i,:      2  i      i i    2  i     i    i
    %
    % where the (x[i], y[i]) are the points along the contour, and let the
    % coefficient vector be
    %
    %                          T
    %      a = [U W V X Y Z] .
    %
    % The algebraic distance between a point and a conic section is the
    % left-hand side of the conic section equation, so that the distance
    % between a point i along the contour and the ellipse described by the
    % vector a is simply D[i,:]a.  We seek the minimum of the sum of
    % squared algebraic distances, which is just ||D a||^2, subject to the
```

```
% constraint ||a||^2 = 1.  We therefore introduce the constrained
% objective function
%
%                 2
%      E = ||D a||  - lambda (||a|| - 1)
%
%            T T               T
%        = a D D a - lambda (a a - 1)
%
% where lambda is a Lagrange multiplier.  The minimum is found
% analytically to occur when
%
%        T
%      D D a = lambda a ,
%
% which is an eigenvalue equation.  The desired coefficient vector, a,
% corresponds to the minimum eigenvalue.
%
% Using the coefficients of the best-fit ellipse, we now calculate
% the beam characteristics.  First, a sign change is applied to the
% coefficients if necessary to force U (and therefore V) to be
% negative.  For convenience, we rewrite the conic section as
%
%      1  T        T
%      - p  A p + B  p + Z = 0 ,
%      2
%
% where p = [x; y],
%
%          / U W \
%      A = |     | ,
%          \ W V /
%
% and B = [X; Y].  We first find the center of the ellipse in order
% to draw it later.  Replacing p with p + p1 in the conic section
% yields
%
%      1   T             T     T
%      - p1  A p1 + (p  A + B ) p1 + Z1 = 0 ,
%      2
%
% where
%
%            1  T        T
%      Z1 = - p  A p + B  p + Z
%            2
%
% is defined for later use.  When p coincides with the center, the
% linear term vanishes; therefore, p solves
%
%      A p + b = 0 .
%
% We next find the roll angle, which is conceptually defined as
% follows, using spherical, not stereographic, coordinates.  If the
% ellipse center is not at boresight, rotate it (and the antenna
% pattern) to boresight along the great circle connecting the two.
% The angle from the great circle with azimuth 0 to the great circle
% along the beam's major axis (direction of maximum width) is the
% roll angle. Alternatively, construct the great circle connecting
% the ellipse center and boresight.  The roll angle is the sum of
% two angles, the angle from the great circle with azimuth 0 to the
% constructed great circle and the angle from the constructed great
% circle to the great circle along the beam's major axis.  Now the
% roll angle so defined is merely the apparent orientation of the
% major axis when viewed in the stereographic projection.  In a
% (stereographic) coordinate system rotated by that angle, the
% off-diagonal element of A (the coefficient W) vanishes; therefore,
% we seek the coordinate system that diagonalizes A.  Replacing p
% with R p2 in the original conic section gives
%
%      1   T   T               T
%      - p2  (R  A R) p2 + (B  R) p2 + Z = 0 .
%      2
%
% The new quadratic coefficient R^T A R will be diagonal if the
% columns of R are the eigenvectors of A.  The new diagonal elements
% U2 and V2 become the eigenvalues, which are explicitly
%
%            U + V    /  2   / U - V \2 \1/2
%      U2 = ----- + | W + | ----- |  |      and
%              2      \      \  2   /  /
%
%            U + V    /  2   / U - V \2 \1/2
%      V2 = ----- - | W + | ----- |  |     .
%              2      \      \  2   /  /
%
% The eigenvalue with the smaller magnitude (U2 above, since U and V
% are negative) corresponds to the major axis.  Therefore, the
% corresponding eigenvector points along the direction of the major
% axis; the other eigenvector, along the minor axis.  The roll angle
% is obtained from the two components of the major axis; explicitly,
% it satisfies
```

```
%
%                      2 W
%     tan (2 roll) = ----- .
%                    U - V
%
% We also use the eigenvectors to draw the ellipse later.  Last, the
% eigenvalues yield the major and minor full widths at half maximum
% of the main beam as
%
%     /    Z1 \1/2        /    Z1 \1/2
%     | -8 -- |      and  | -8 -- |    ,
%     \    U2 /           \    V2 /
%
% respectively.  As these were derived in the stereographic
% projection, a factor of (1 + cos polar) is applied to obtain the
% approximate widths in real angles.
%
if capClosed
  %
  % Construct the contour
  %
  capIndx = beamIndx (gSqr (beamIndx) >= beamWidLvl);  % elements at or above level
  capBool = logical (zeros (ay, ax));
  capBool (capIndx) = ones (size (capIndx));
  adjc = [ay ay+1 1 -ay+1 -ay -ay-1 -1 ay-1];  % clockwise in matrix row-column coordinates
  dirIndx = 1;                                 % initial index into adjc
  intIndx = capIndx (1);                       % initial index of interpolation center
  dirIndxSt = 0;                               % get loop started
  intIndxSt = 0;
  capContX = [];                               % empty contour coordinates
  capContY = [];
  while capBool (intIndx + adjc (dirIndx))      % next element is inside cap?
    intIndx = intIndx + adjc (dirIndx);        % keep moving until edge is reached
  end
  while (intIndx ~= intIndxSt) | (dirIndx ~= dirIndxSt)  % back at starting point?
    adjcInc = abs (adjc (dirIndx));                      % no; get magnitude
    if (adjcInc == 1) | (adjcInc == ay)                  % looking across row or column?
      if adjcInc == ay                                   % across column?
        capContY1 = dirCosYMtx (intIndx);                % yes; interpolate in x
        capContX1 = dirCosXMtx (intIndx) + (beamWidLvl - gSqr (intIndx)) ...
          * (dirCosXMtx (intIndx + adjc (dirIndx)) - dirCosXMtx (intIndx)) ...
          / (gSqr       (intIndx + adjc (dirIndx)) - gSqr       (intIndx));
      else                                               % across row
        capContX1 = dirCosXMtx (intIndx);                % interpolate in y
        capContY1 = dirCosYMtx (intIndx) + (beamWidLvl - gSqr (intIndx)) ...
          * (dirCosYMtx (intIndx + adjc (dirIndx)) - dirCosYMtx (intIndx)) ...
          / (gSqr       (intIndx + adjc (dirIndx)) - gSqr       (intIndx));
      end
      if isempty (capContX)                              % first point?
        capContX = capContX1;                            % yes; store it
        capContY = capContY1;
        intIndxSt = intIndx;                             % remember starting point
        dirIndxSt = dirIndx;
      elseif (capContX (length (capContX)) ~= capContX1) ...  % duplicate? (e.g.,
           & (capContY (length (capContY)) ~= capContY1)      %   element equals level)
        capContX = [capContX; capContX1];                % no; append it
        capContY = [capContY; capContY1];
      end
    end                                                  % (no diagonal interpolation)
    dirIndx = dirIndx + 1;                               % next direction
    if dirIndx > length (adjc)                           % cycle
      dirIndx = 1;
    end
    adjcInc = abs (adjc (dirIndx));                      % no; get magnitude of step
    if capBool (intIndx + adjc (dirIndx))                % next element is inside cap?
      intIndx = intIndx + adjc (dirIndx);                % yes; becomes new interpolation center
      dirIndx = dirIndx - length (adjc) / 2 + 1;         % reverse, then ahead one increment
      if (adjcInc == 1) | (adjcInc == ay)                % stepped in row or column?
        dirIndx = dirIndx + 1;                           % yes; ahead an extra increment
      end                                                %   (useless to look back diagonally)
      if dirIndx < 1                                     % cycle
        dirIndx = dirIndx + length (adjc);
      end
    end
  end % while                                            % contour complete
  %
  % Fit an ellipse in stereographic coordinates
  %
  capContZ = sqrt (1 - capContX.^2 - capContY.^2);
  capContXS = capContX ./ (1 + capContZ);                      % to stereographic coords
  capContYS = capContY ./ (1 + capContZ);
  cDesMtx = [capContXS.*capContXS/2 capContXS.*capContYS ...   % to form design matrix
    capContYS.*capContYS/2 capContXS ...                       %   for least-squares fit
    capContYS ones(size(capContXS))];
  cDesMtx = cDesMtx' * cDesMtx;                                % done
  capContX = capContX ([1:length(capContX) 1]);                % close contour for plotting
  capContY = capContY ([1:length(capContY) 1]);
  capContZ = capContZ ([1:length(capContZ) 1]);
  [cEigVec, cEigVal] = eig (cDesMtx);                          % eigenvectors and -values
  [cEigValMin, cEigValMinIdx] = min (diag (cEigVal));          % minimum eigenvalue
  cPoly = cEigVec (:, cEigValMinIdx);                          %   and matching vector
  cPoly = -cPoly * sign (cPoly (1));                           % to have negative eigenvalues below
```

```
    %
    % Obtain beam characteristics from ellipse coefficients
    %
    cHessMtx = [cPoly(1) cPoly(2)                            % Hessian matrix;
                cPoly(2) cPoly(3)];                          %   second derivatives
    cDervMtx = [cPoly(4)                                     % first derivative matrix
                cPoly(5)];
    capCenter = -cHessMtx \ cDervMtx;                        % ellipse center
    capConst = cPoly (6) + cDervMtx' * capCenter / 2;        % new constant coefficient
    [cEigVec, cEigVal] = eig (cHessMtx);
    [cEigVal, cEigValOrd] = sort (diag (cEigVal));           % ascending order
    cEigVec = cEigVec (:, cEigValOrd);                       % corresponding order
    if cEigVec (1, 2) == 0                                   % special case?
      roll = pi / 2;
    else
      roll = atan (cEigVec (2, 2) / cEigVec (1, 2));         % angle to major axis
    end
    roll = roll - pi/2 + azmthOffst;                         % make -pi/2 <
    roll = roll - ceil (roll / pi) * pi + pi/2 - azmthOffst; %   roll + azmthOffst <= pi/2
    hpbw = sqrt (-8 * capConst ./ cEigVal);                  % two-vector
    %
    % Construct the fitted ellipse for plotting (in stereographic coordinates)
    %
    theta = (0 : 10 / 360 : 1) * twopi;
    cFit = capCenter * ones (size (theta)) ...  % calculate some
      + 0.5 * (cEigVec * diag (hpbw) ...        %   points along the
             * [sin(theta); cos(theta)]);       %   fitted ellipse
    cFitR2 = sum (cFit.^2);                     % squared radius in stereographic coords
    cFitZ = (1 - cFitR2) ./ (1 + cFitR2);       % z direction cosine
    cFitX = cFit (1, :) .* (1 + cFitZ);         % undo stereographic projection
    cFitY = cFit (2, :) .* (1 + cFitZ);
    hpbw = hpbw * (1 + pz);                     % undo widths, too
    hpbwMjr = hpbw (2);
    hpbwMnr = hpbw (1);
  else
    roll = nan;
    hpbwMjr = nan;
    hpbwMnr = nan;
  end
  clear capBool adjc dirIndx intIndx dirIndxSt intIndxSt adjcInc capContX1 capContY1;
  clear capContXS capContYS cDesMtx cEigVec cEigVal cEigValMin cEigValMinIdx cPoly;
  clear cHessMtx cDervMtx capCenter capConst cEigValOrd hpbw;
  clear theta cFit cFitR2;

  % Calculate power in visible space, main beam, and sidelobes; main
  % beam and sidelobe solid angles; and average sidelobe level
  %
  % These calculations involve integrals over the hemisphere or portions
  % of it.  The integrals are carried out in direction cosine
  % coordinates by multiplying the integrand by the appropriate
  % Jacobian.
  %
  % The integrals are evaluated using the midpoint approximation, for
  % which the starting point is the Taylor series expansion of g (x, y)
  % to second order:
  %
  %      g (a + u, b + v)
  %
  %             /                   1  2                1  2    \
  %           = | g + u g  + v g  + - u g   + u v g   + - v g   |   .
  %             \        x      y   2    xx         xy  2    yy /a,b
  %
  % Then
  %
  %      / c/2 / d/2
  %      | du  | dv  g (a + u, b + v) =
  %      /-c/2 /-d/2
  %
  %             /         1        2      2     \
  %           = | c d g + -- c d (c g  + d g  ) |   .
  %             \         24         xx     yy  /a,b
  %
  % The midpoint approximation keeps the first term and neglects the
  % quadratic terms.  The maximum amount neglected is
  %
  %      c d /  2 |         |    2 |         | \
  %      --- | c  |g  (a, b)| + d  |g  (a, b)| | ,
  %      24  \    | xx      |      | yy      | /
  %
  % which we use as the error estimate for each interior point of the
  % integration, approximating the second derivatives with scaled second
  % differences.
  %
  % To the interior error is added an estimate of the error due to
  % finite sampling at the integral limits; the estimate is half the
  % value of the integrand at the outermost samples.
  %
  % (The error estimate calculations have been commented out for
  % speed.)
  %
% visbEdgeIndx = visBool;
% visbEdgeIndx (2 : ay - 1, 2 : ax - 1) = ...
```

```
%   ( visbEdgeIndx (2 : ay - 1, 3 : ax    ) & visbEdgeIndx (1 : ay - 2, 2 : ax - 1) ...
%   & visbEdgeIndx (2 : ay - 1, 1 : ax - 2) & visbEdgeIndx (3 : ay    , 2 : ax - 1) );
% visbEdgeIndx = find (visBool - visbEdgeIndx);
% beamEdgeIndx = zeros (ay, ax);
% beamEdgeIndx (beamIndx) = ones (size (beamIndx));
% sideEdgeIndx = beamEdgeIndx;
% beamEdgeIndx (2 : ay - 1, 2 : ax - 1) = ...
%   ( beamEdgeIndx (2 : ay - 1, 3 : ax    ) & beamEdgeIndx (1 : ay - 2, 2 : ax - 1) ...
%   & beamEdgeIndx (2 : ay - 1, 1 : ax - 2) & beamEdgeIndx (3 : ay    , 2 : ax - 1) );
% beamEdgeIndx (beamIndx) = 1 - beamEdgeIndx (beamIndx);
% beamEdgeIndx = find (beamEdgeIndx);
% sideEdgeIndx (2 : ay - 1, 2 : ax - 1) = ...
%   ( sideEdgeIndx (2 : ay - 1, 3 : ax    ) | sideEdgeIndx (1 : ay - 2, 2 : ax - 1) ...
%   | sideEdgeIndx (2 : ay - 1, 1 : ax - 2) | sideEdgeIndx (3 : ay    , 2 : ax - 1) );
% sideEdgeIndx (beamIndx) = zeros (size (beamIndx));
% sideEdgeIndx = find (sideEdgeIndx);
  areaFact = zeros (ay, ax);                              % areal factor for integrating:
  areaFact (visBool) = ...                                %   Jacobian [1 / cos polar]
    1 ./ (txLim * tyLim * dirCosZMtx (visBool));          %   and grid spacing
  areaFactLim = 2 / sqrt (txLim * tyLim);                 % edge points may exceed this
  tooBig = find (areaFact > areaFactLim);                 %   arbitrary limit; force those
  areaFact (tooBig) = ones (size (tooBig)) * areaFactLim; %   that do to comply
% sldAng = sum (areaFact (:));
% sldAngErrRel = abs (sldAng / (2 * pi) - 1);
% areaFactUnc = zeros (ay, ax);
% areaFactUnc (2 : ay - 1, 2 : ax - 1) = ...              % error estimate based on second-
%   (  abs (diff (areaFact (2 : ay - 1, :)', 2)') ...     %   order Taylor expansion
%    + abs (diff (areaFact (:, 2 : ax - 1) , 2) ) ) / 24;
% areaFactUnc ([1 ay], :) = areaFactUnc ([2 ay-1], :);    % assume errors at outer edge equal
% areaFactUnc (:, [1 ax]) = areaFactUnc (:, [2 ax-1]);    %   those of nearest neighbors
  intgrnd = areaFact .* gSqr;                             % to integrate gSqr
% intgrndUnc = zeros (ay, ax);                            % error estimate for integrand
% intgrndUnc (2 : ay - 1, 2 : ax - 1) = ...
%   (  abs (diff (intgrnd (2 : ay - 1, :)', 2)') ...
%    + abs (diff (intgrnd (:, 2 : ax - 1) , 2) ) ) / 24;
% intgrndUnc ([1 ay], :) = intgrndUnc ([2 ay-1], :);      % assume errors at outer edge equal
% intgrndUnc (:, [1 ax]) = intgrndUnc (:, [2 ax-1]);      %   those of nearest neighbors
  powerVisb = sum (intgrnd (:));
% powerVisbUnc = sum (intgrndUnc (:));
  directivityDB = 10 * log10 (4 * pi * gSqrMax / powerVisb);
  if beamExist
%   sldAngMain = sum (areaFact (beamIndx));               % main beam solid angle
%   sldAngMainUnc = sum (areaFactUnc (beamIndx)) + sum (areaFact (beamEdgeIndx)) / 2;
    powerMain = sum (intgrnd (beamIndx));                 % power in the main beam
%   powerMainUnc = sum (intgrndUnc (beamIndx)) + sum (intgrnd (beamEdgeIndx)) / 2;
    powerSide = sum (intgrnd (sideBool));                 % power in the sidelobes
%   powerSideUnc = sum (intgrndUnc (:)) - sum (intgrndUnc (beamIndx))...
%     + sum (intgrnd (sideEdgeIndx)) / 2;
    sldAngSide = sum (areaFact (sideBool));               % sidelobe equivalent solid angle
%   sldAngSideUnc = sum (areaFactUnc (sideBool)) - sum (areaFactUnc (beamIndx)) ...
%     + sum (areaFact (sideEdgeIndx)) / 2;
  else
%   sldAngMain = nan;
%   sldAngMainUnc = nan;
    powerMain = nan;
%   powerMainUnc = nan;
    powerSide = nan;
%   powerSideUnc = nan;
    sldAngSide = nan;
%   sldAngSideUnc = nan;
  end
  powerMainVisbDB = 10 * log10 (powerMain / powerVisb);   % ratio of main beam to visible power
% powerMainVisbUnc = powerMainUnc / powerVisb ...
%   + powerMain * powerVisbUnc / powerVisb^2;
  powerVisbSideDB = 10 * log10 (powerVisb / powerSide);   % ratio of visible to sidelobe power
% powerVisbSideUnc = powerVisbUnc / powerSide ...
%   + powerVisb * powerSideUnc / powerSide^2;
  powerMainSideDB = 10 * log10 (powerMain / powerSide);   % ratio of main beam to sidelobe power
% powerMainSideUnc = powerMainUnc / powerSide ...
%   + powerMain * powerSideUnc / powerSide^2;
  powerSideAvgDB = 10 * log10 (powerSide / (sldAngSide * gSqrMax));  % average sidelobe power
% powerSideAvgUnc = powerSideUnc / sldAngSide ...
%   + powerSide * sldAngSideUnc / sldAngSide^2;
  clear intgrnd areaFact areaFactLim tooBig;
% clear intgrndUnc areaFactUnc;

  % Locate nearest and largest sidelobes
  %
  % We wish to identify the sidelobe closest in angle to the main beam
  % and the sidelobe with the largest peak power.  We first find the
  % local maxima in the sidelobe region, then for each we determine
  % the possible ranges for actual distance from the main beam and
  % peak power.  (The uncertainties arise from the discrete sampling
  % of the array factor.)  Using the ranges we select those peaks that
  % could possibly be the closest or largest.  For each of these
  % candidates a more precise location and peak power is computed by
  % interpolating over neighboring data.  Finally, based on these
  % results, the closest and nearest sidelobes are identified.
  %
  if peakVisb
    %
    % Find local maxima (sidelobe peaks), using discrete differences
```

```
% to approximate derivatives.  The differences are formed from the
% magnitude of the array factor, not the squared magnitude; since
% the behavior should already be parabolic near peaks, squaring
% would produce fourth-order behavior and make second-order
% interpolation less accurate.
%
% Key to the variables below:
%   X   first differences in x
%   Y   first differences in y
%   XX  second differences in x
%   YY  second differences in y
%   X2  first differences in x with double step
%   XY  cross differences in x and y
%   XC  nonzero where first difference in x changes sign
%   YC  nonzero where first difference in y changes sign
%
gMagX  = gMag (:, 2:ax) - gMag (:, 1:ax-1);
gMagY  = gMag (2:ay, :) - gMag (1:ay-1, :);
gMagXX = [zeros(ay,1)  (gMagX (:, 2:ax-1)   - gMagX (:, 1:ax-2))      zeros(ay,1)];
gMagYY = [zeros(1,ax); (gMagY (2:ay-1, :)   - gMagY (1:ay-2, :));     zeros(1,ax)];
gMagX2 = [zeros(ay,1)  (gMag  (:, 3:ax  )   - gMag  (:, 1:ax-2))/2    zeros(ay,1)];
gMagXY = [zeros(1,ax); (gMagX2(3:ay  , :)   - gMagX2(1:ay-2, :))/2;   zeros(1,ax)];
gMagXC = [zeros(ay,1)  (gMagX (:, 2:ax-1) .* gMagX (:, 1:ax-2) < 0)  zeros(ay,1)];
gMagYC = [zeros(1,ax); (gMagY (2:ay-1, :) .* gMagY (1:ay-2, :) < 0); zeros(1,ax)];
slIndx = find ( sideBool ...                    % identify sidelobe points where
  &  gMagXC      &  gMagYC       ...            %   first derivatives change sign,
  & (gMagXX < 0) & (gMagYY < 0) ...             %   second derivatives are negative, and
  & (gMagXY.^2 - gMagXX .* gMagYY < 0) );       %   discriminant is negative
if isempty (slIndx)                             % none found?
  slIndx = [];
  slNrstDist = nan;
  slNrstPowrDB = nan;
  slNrstVec = nan * ones (1, 3);
  slLgstDist = nan;
  slLgstPowrDB = nan;
  slLgstVec = nan * ones (1, 3);
else                                            % found some peaks
  %
  % Compute possible ranges of distances and powers; identify
  % candidates for closest and largest peaks
  %
  toward = sign (px - dirCosXMtx (slIndx)) * ay ...   % index increment to neighbor
         + sign (py - dirCosYMtx (slIndx));           %   closer to pointing vector
  slCosDistMax = ...                                  % cosine of maximum possible
      px * dirCosXMtx (slIndx + toward) ...           %   angle between pointing
    + py * dirCosYMtx (slIndx + toward) ...           %   vector and each peak;
    + pz * dirCosZMtx (slIndx + toward);              %   dot product
  slCosDistMin = ...                                  % likewise for minimum
      px * dirCosXMtx (slIndx - toward) ...           %   possible angle
    + py * dirCosYMtx (slIndx - toward) ...
    + pz * dirCosZMtx (slIndx - toward);
  slNrstBool = (slCosDistMax >= max (slCosDistMin)); % true if peak might be the closest
  slPowr = (gMag (slIndx) ...                         % estimate largest possible
    + ( -gMagXX (slIndx) ...                          %   interpolated power by adding
        + 2 * abs (gMagXY (slIndx)) ...               %   an error estimate based on the
        - gMagYY (slIndx) ) / 8).^2;                  %   differences computed above
  slLgstBool = (slPowr >= max (gSqr (slIndx)));       % true if peak might be the largest
  slCandIndx = slIndx (slNrstBool | slLgstBool);      % candidates for closest and largest
  numCand = length (slCandIndx);                      % number of candidates
  %
  % Interpolate powers and locations for candidates
  %
  slx = zeros (numCand, 1);                           % allocate space for x and
  sly = zeros (numCand, 1);                           %   y direction cosines
  slPowr = zeros (numCand, 1);                        %   and powers
  for slPtr = 1 : numCand                             % loop through candidates
    slCol = ceil (slCandIndx (slPtr) / ay);           % separate index into column
    slRow = slCandIndx (slPtr) - (slCol - 1) * ay;    %   and row indices
    intRad = 1;                                       % interpolation radius
    slOK = 0;                                         % initialize
    while ~slOK & intRad < 3                          % no answer yet but too early to bail?
      slNbrs = [-intRad : intRad];                    % offsets to neighbors
      slxFit = dirCosXMtx (slRow + slNbrs, slCol + slNbrs);  % x, y, and z coordinates of
      slyFit = dirCosYMtx (slRow + slNbrs, slCol + slNbrs);  %   neighbors (using magnitude,
      slzFit = gMag       (slRow + slNbrs, slCol + slNbrs);  %   not power, for z)
      slxFit = slxFit (:);
      slyFit = slyFit (:);
      slzFit = slzFit (:);
      slDesMtx = [slxFit.*slxFit/2 slxFit.*slyFit slyFit.*slyFit/2 ...  % design matrix
        slxFit slyFit ones(size(slxFit))];
      [Q, R] = qr (slDesMtx, 0);                      % now slDesMtx = Q * R; Q' * Q = eye
      slPoly = R \ (Q' * slzFit);                     % solves slDesMtx * slPoly = slzFit
      slVec = -[slPoly(1) slPoly(2); slPoly(2) slPoly(3)] ...  % find critical point
        \ [slPoly(4); slPoly(5)];
      slOK = ...                                      % is interpolated point
          (slVec (1) > dirCosX (slCol - intRad)) ...  %   inside neighborhood?
        & (slVec (1) < dirCosX (slCol + intRad)) ...  %   (pathological cases can
        & (slVec (2) > dirCosY (slRow - intRad)) ...  %   place it outside)
        & (slVec (2) < dirCosY (slRow + intRad));
      if ~slOK                                        % outside neighborhood?
        intRad = intRad + 1;                          % yes; cast a wider net
      end
```

```
          end                                           % end interpolation attempts
          if slOK                                       % interpolation successful?
            slx (slPtr) = slVec (1);                    % yes; keep interpolated point
            sly (slPtr) = slVec (2);
            slPowr (slPtr) = ...                        % interpolate power
              ([slx(slPtr).*slx(slPtr)/2 slx(slPtr).*sly(slPtr) ...
                sly(slPtr).*sly(slPtr)/2 slx(slPtr) sly(slPtr) 1] * slPoly).^2;
          else                                          % interpolation failed
            slx (slPtr) = dirCosXMtx (slCandIndx (slPtr));  % use grid location of
            sly (slPtr) = dirCosYMtx (slCandIndx (slPtr));  %   sampled maximum
            slPowr (slPtr) = gSqr (slCandIndx (slPtr));     % use sampled power
          end
        end                                             % end of loop through candidates
        %
        % Select closest and largest peaks
        %
        slz = sqrt (1 - slx.^2 - sly.^2);               % z direction cosines
        slDist = acos (px * slx + py * sly + pz * slz);  % angular distances
        [slNrstDist, slNrstIndx] = min (slDist);                    % smallest distance
        slNrstPowrDB = 10 * log10 (slPowr (slNrstIndx) / gSqrMax);  %   and corresponding power
        slNrstVec = [slx(slNrstIndx) sly(slNrstIndx) slz(slNrstIndx)];  % keep vector for plotting
        [slLgstPowrDB, slLgstIndx] = max (slPowr);                  % largest power
        slLgstPowrDB = 10 * log10 (slLgstPowrDB / gSqrMax);         %   converted to dB
        slLgstDist = slDist (slLgstIndx);                           % and corresponding distance
        slLgstVec = [slx(slLgstIndx) sly(slLgstIndx) slz(slLgstIndx)];  % keep vector
      end
    else                                                % peak is invisible
      slIndx = [];
      slNrstDist = nan;
      slNrstPowrDB = nan;
      slNrstVec = nan * ones (1, 3);
      slLgstDist = nan;
      slLgstPowrDB = nan;
      slLgstVec = nan * ones (1, 3);
    end
    clear gMagX gMagY gMagXX gMagYY gMagX2 gMagXY gMagXC gMagYC;
    clear toward slCosDistMax slCosDistMin;
    clear slNrstBool slLgstBool slCandIndx numCand;
    clear slx sly slz slPowr slPtr slCol slRow intRad slOK;
    clear slNbrs slxFit slyFit slzFit slDesMtx Q R slPoly slVec;
    clear slDist slnrstindx slLgstIndx;
    %
    % Record characteristics
    %
    % In order to calculate running means and standard deviations of n
    % realizations, we accumulate the mean and variance
    %
    %             1  n
    %       M  = - sum x    and
    %        n   n m=1  m
    %
    %               1    n            2
    %       V  = ----- sum (x  - M )
    %        n   n - 1 m=1   m    n
    %
    % using the updating formulas
    %
    %                       x  - M
    %                        n    n-1
    %       M  = M     + ---------   and
    %        n    n-1          n
    %
    %              n - 2         1            2
    %       V  = ----- V     + - (x  - M    )  .
    %        n   n - 1  n-1   n   n    n-1
    %
    % The running mean is simply M[n], and the standard deviation is
    % sqrt (V[n]).  M is accumulated in the first column of a matrix; V
    % in the second.  This method is more immune to roundoff error than
    % accumulating the sums of values and squares [1].
    %
    % If the beam is invisible, the excitations and array factor are
    % saved to an automatically-named file.
    %
    % [1] N. J. Higham, _Accuracy and Stability of Numerical Algorithms_.
    % Philadelphia, PA: SIAM, 1996, pp. 12-13.
    %
    if beamVisb | (numRlz == 1)
      if isnan (numAcc (indx))
        numAcc (indx) = 1;
        pxS                 (indx, 1) = px;
        pxS                 (indx, 2) = 0;
        pyS                 (indx, 1) = py;
        pyS                 (indx, 2) = 0;
        pzS                 (indx, 1) = pz;
        pzS                 (indx, 2) = 0;
        errPointS           (indx, 1) = errPoint;
        errPointS           (indx, 2) = 0;
      % errPointUncS        (indx, 1) = errPointUnc;
      % errPointUncS        (indx, 2) = 0;
        beamPowerDBS        (indx, 1) = beamPowerDB;
        beamPowerDBS        (indx, 2) = 0;
```

```
            beamDepthDBS         (indx, 1) = beamDepthDB;
            beamDepthDBS         (indx, 2) = 0;
            hpbwMjrS             (indx, 1) = hpbwMjr;
            hpbwMjrS             (indx, 2) = 0;
            hpbwMnrS             (indx, 1) = hpbwMnr;
            hpbwMnrS             (indx, 2) = 0;
            rollS                (indx, 1) = roll;
            rollS                (indx, 2) = 0;
            directivityDBS       (indx, 1) = directivityDB;
            directivityDBS       (indx, 2) = 0;
            powerMainVisbDBS     (indx, 1) = powerMainVisbDB;
            powerMainVisbDBS     (indx, 2) = 0;
            powerMainSideDBS     (indx, 1) = powerMainSideDB;
            powerMainSideDBS     (indx, 2) = 0;
            powerVisbSideDBS     (indx, 1) = powerVisbSideDB;
            powerVisbSideDBS     (indx, 2) = 0;
            powerSideAvgDBS      (indx, 1) = powerSideAvgDB;
            powerSideAvgDBS      (indx, 2) = 0;
            slNrstDistS          (indx, 1) = slNrstDist;
            slNrstDistS          (indx, 2) = 0;
            slNrstPowrDBS        (indx, 1) = slNrstPowrDB;
            slNrstPowrDBS        (indx, 2) = 0;
            slLgstDistS          (indx, 1) = slLgstDist;
            slLgstDistS          (indx, 2) = 0;
            slLgstPowrDBS        (indx, 1) = slLgstPowrDB;
            slLgstPowrDBS        (indx, 2) = 0;
          else
            numAcc (indx) = numAcc (indx) + 1;
            factV = (numAcc (indx) - 2) / (numAcc (indx) - 1);
            dev = px - pxS (indx, 1);
            pxS              (indx, 2) = pxS              (indx, 2) * factV + dev^2 / numAcc (indx);
            pxS              (indx, 1) = pxS              (indx, 1)         + dev   / numAcc (indx);
            dev = py - pyS (indx, 1);
            pyS              (indx, 2) = pyS              (indx, 2) * factV + dev^2 / numAcc (indx);
            pyS              (indx, 1) = pyS              (indx, 1)         + dev   / numAcc (indx);
            dev = pz - pzS (indx, 1);
            pzS              (indx, 2) = pzS              (indx, 2) * factV + dev^2 / numAcc (indx);
            pzS              (indx, 1) = pzS              (indx, 1)         + dev   / numAcc (indx);
            dev = errPoint - errPointS (indx, 1);
            errPointS        (indx, 2) = errPointS        (indx, 2) * factV + dev^2 / numAcc (indx);
            errPointS        (indx, 1) = errPointS        (indx, 1)         + dev   / numAcc (indx);
          % dev = errPointUnc - errPointUncS (indx, 1);
          % errPointUncS     (indx, 2) = errPointUncS     (indx, 2) * factV + dev^2 / numAcc (indx);
          % errPointUncS     (indx, 1) = errPointUncS     (indx, 1)         + dev   / numAcc (indx);
            dev = beamPowerDB - beamPowerDBS (indx, 1);
            beamPowerDBS     (indx, 2) = beamPowerDBS     (indx, 2) * factV + dev^2 / numAcc (indx);
            beamPowerDBS     (indx, 1) = beamPowerDBS     (indx, 1)         + dev   / numAcc (indx);
            dev = beamDepthDB - beamDepthDBS (indx, 1);
            beamDepthDBS     (indx, 2) = beamDepthDBS     (indx, 2) * factV + dev^2 / numAcc (indx);
            beamDepthDBS     (indx, 1) = beamDepthDBS     (indx, 1)         + dev   / numAcc (indx);
            dev = hpbwMjr - hpbwMjrS (indx, 1);
            hpbwMjrS         (indx, 2) = hpbwMjrS         (indx, 2) * factV + dev^2 / numAcc (indx);
            hpbwMjrS         (indx, 1) = hpbwMjrS         (indx, 1)         + dev   / numAcc (indx);
            dev = hpbwMnr - hpbwMnrS (indx, 1);
            hpbwMnrS         (indx, 2) = hpbwMnrS         (indx, 2) * factV + dev^2 / numAcc (indx);
            hpbwMnrS         (indx, 1) = hpbwMnrS         (indx, 1)         + dev   / numAcc (indx);
            dev = roll - rollS (indx, 1);
            rollS            (indx, 2) = rollS            (indx, 2) * factV + dev^2 / numAcc (indx);
            rollS            (indx, 1) = rollS            (indx, 1)         + dev   / numAcc (indx);
            dev = directivityDB - directivityDBS (indx, 1);
            directivityDBS   (indx, 2) = directivityDBS   (indx, 2) * factV + dev^2 / numAcc (indx);
            directivityDBS   (indx, 1) = directivityDBS   (indx, 1)         + dev   / numAcc (indx);
            dev = powerMainVisbDB - powerMainVisbDBS (indx, 1);
            powerMainVisbDBS (indx, 2) = powerMainVisbDBS (indx, 2) * factV + dev^2 / numAcc (indx);
            powerMainVisbDBS (indx, 1) = powerMainVisbDBS (indx, 1)         + dev   / numAcc (indx);
            dev = powerMainSideDB - powerMainSideDBS (indx, 1);
            powerMainSideDBS (indx, 2) = powerMainSideDBS (indx, 2) * factV + dev^2 / numAcc (indx);
            powerMainSideDBS (indx, 1) = powerMainSideDBS (indx, 1)         + dev   / numAcc (indx);
            dev = powerVisbSideDB - powerVisbSideDBS (indx, 1);
            powerVisbSideDBS (indx, 2) = powerVisbSideDBS (indx, 2) * factV + dev^2 / numAcc (indx);
            powerVisbSideDBS (indx, 1) = powerVisbSideDBS (indx, 1)         + dev   / numAcc (indx);
            dev = powerSideAvgDB - powerSideAvgDBS (indx, 1);
            powerSideAvgDBS  (indx, 2) = powerSideAvgDBS  (indx, 2) * factV + dev^2 / numAcc (indx);
            powerSideAvgDBS  (indx, 1) = powerSideAvgDBS  (indx, 1)         + dev   / numAcc (indx);
            dev = slNrstDist - slNrstDistS (indx, 1);
            slNrstDistS      (indx, 2) = slNrstDistS      (indx, 2) * factV + dev^2 / numAcc (indx);
            slNrstDistS      (indx, 1) = slNrstDistS      (indx, 1)         + dev   / numAcc (indx);
            dev = slNrstPowrDB - slNrstPowrDBS (indx, 1);
            slNrstPowrDBS    (indx, 2) = slNrstPowrDBS    (indx, 2) * factV + dev^2 / numAcc (indx);
            slNrstPowrDBS    (indx, 1) = slNrstPowrDBS    (indx, 1)         + dev   / numAcc (indx);
            dev = slLgstDist - slLgstDistS (indx, 1);
            slLgstDistS      (indx, 2) = slLgstDistS      (indx, 2) * factV + dev^2 / numAcc (indx);
            slLgstDistS      (indx, 1) = slLgstDistS      (indx, 1)         + dev   / numAcc (indx);
            dev = slLgstPowrDB - slLgstPowrDBS (indx, 1);
            slLgstPowrDBS    (indx, 2) = slLgstPowrDBS    (indx, 2) * factV + dev^2 / numAcc (indx);
            slLgstPowrDBS    (indx, 1) = slLgstPowrDBS    (indx, 1)         + dev   / numAcc (indx);
          end
        else
          eval (['save case' sprintf('%0.0f',indx) '-' sprintf('%0.0f',rlzNum) ' exc gSqr']);
        end
        clear factV dev
```

```
    % Calculate spherical coordinates of average pointing vector
    %
    % Let <px>, <py>, and <pz> denote the average direction cosines of
    % the pointing vectors, and let vx, vy, and vz denote the
    % corresponding variances.  We wish to express the direction of the
    % average pointing vector [<px> <py> <pz>] in spherical coordinates.
    % First, note that the average pointing vector has the norm
    %
    %                  2        2        2
    %      p = (<px>  + <py>  + <pz> ) ,
    %
    % which is less than one if not all realizations are colinear.  The
    % spherical angles are then given by
    %
    %                         1       2        2 1/2
    %      sin pointPolar = - (<px>  + <py> )      and
    %                         p
    %
    %                       <py>
    %      tan pointAzmth = ---- ,
    %                       <px>
    %
    % where we use sin pointPolar instead of the cosine for accuracy
    % near boresight.
    %
    % Also, we wish to estimate the rms angular deviation of a
    % realization of the pointing vector from the mean.  Adopting an
    % unsophisticated method, we add the variances vx and vy to obtain
    % an equivalent area in the x-y direction cosine plane, then divide
    % the area by cos PointPolar to yield a solid angle on the
    % hemisphere.  The square root of that area yields the rms angular
    % deviation.
    %
    radSqr = pxS (indx, 1)^2 + pyS (indx, 1)^2;
    pointPolar = asin (sqrt (radSqr / (radSqr + pzS (indx, 1)^2)));
    pointAzmth = atan2 (pyS (indx, 1), pxS (indx, 1));
    if isnan (pointPolar)
      pointStdDev = nan;
    else
      pointStdDev = sqrt ((pxS (indx, 2) + pyS (indx, 2)) / cos (pointPolar));
    end
    clear radSqr

    % Print performance characteristics
    %
    % Generally, the following statements print the results of the
    % analysis followed by the standard deviations of each result in
    % curly brackets.
    %
    if 1  % INPUT 0 to suppress output, 1 to print
      fprintf (1, '\nMeans and [std devs] for %0.0f of %0.0f realizations\n', rlzNum, numRlz);
      fprintf (1, 'beam direction : (%0.3f, %0.3f) deg, std dev %0.3f deg\n', ...
        pointPolar / rpd, (pointAzmth + azmthOffst) / rpd, pointStdDev / rpd);
      fprintf (1, 'pointing error : %8.4f [%0.4f] deg\n', ...
        errPointS (indx, 1) / rpd, sqrt (errPointS (indx, 2)) / rpd);
    % fprintf (1, 'pntng error unc: %8.4f [%0.4f] deg (2 sigma)\n', ...
    %   2 * errPointUncS (indx, 1) / rpd, 2 * sqrt (errPointUncS (indx, 2)) / rpd);
      fprintf (1, 'peak power dens: %7.3f  [%0.3f ] dB\n', ...
        beamPowerDBS (indx, 1), sqrt (beamPowerDBS (indx, 2)));
      fprintf (1, 'beam depth     : %6.2f   [%0.2f  ] dB re peak\n', ...
        beamDepthDBS (indx, 1), sqrt (beamDepthDBS (indx, 2)));
      fprintf (1, 'beam width     : (%6.3f  [%0.3f ], %0.3f [%0.3f]) deg\n', ...
        hpbwMjrS (indx, 1) / rpd, sqrt (hpbwMjrS (indx, 2)) / rpd, ...
        hpbwMnrS (indx, 1) / rpd, sqrt (hpbwMnrS (indx, 2)) / rpd);
      fprintf (1, 'beam roll      : %6.2f   [%0.2f  ] deg\n', ...
        (rollS (indx, 1) + azmthOffst) / rpd, sqrt (rollS (indx, 2)) / rpd);
      fprintf (1, 'directivity    : %7.3f  [%0.3f ] dB\n', ...
        directivityDBS (indx, 1), sqrt (directivityDBS (indx, 2)));
      fprintf (1, 'power ratio m/v: %7.3f  [%0.3f ] dB\n', ...
        powerMainVisbDBS (indx, 1), sqrt (powerMainVisbDBS (indx, 2)));
      fprintf (1, 'power ratio m/s: %7.3f  [%0.3f ] dB\n', ...
        powerMainSideDBS (indx, 1), sqrt (powerMainSideDBS (indx, 2)));
      fprintf (1, 'power ratio v/s: %7.3f  [%0.3f ] dB\n', ...
        powerVisbSideDBS (indx, 1), sqrt (powerVisbSideDBS (indx, 2)));
      fprintf (1, 'avg sidelobe   : %6.2f   [%0.2f  ] dB re peak\n', ...
        powerSideAvgDBS (indx, 1), sqrt (powerSideAvgDBS (indx, 2)));
      fprintf (1, 'nrst sidelobe  : %6.2f   [%0.2f] dB re peak, %0.2f [%0.2f] deg off beam\n', ...
        slNrstPowrDBS (indx, 1), sqrt (slNrstPowrDBS (indx, 2)), ...
        slNrstDistS (indx, 1) / rpd, sqrt (slNrstDistS (indx, 2)) / rpd);
      fprintf (1, 'lgst sidelobe  : %6.2f   [%0.2f] dB re peak, %0.2f [%0.2f] deg off beam\n', ...
        slLgstPowrDBS (indx, 1), sqrt (slLgstPowrDBS (indx, 2)), ...
        slLgstDistS (indx, 1) / rpd, sqrt (slLgstDistS (indx, 2)) / rpd);
    end

  end  % loop over realizations

  % Plot performance characteristics as function of independent variable
  % (summary plot).
  %
  % If more than one realization has been accumulated for any value of
  % the independent variable, the mean is plotted with uncertainty bars.
  % The extension of the uncertainty bar above the mean equals one
```

```
% standard deviation, and likewise below the mean.  Otherwise, only
% the values for the one realization are plotted.
%
% Plots that show more than one measure distinguish them by color or
% line style and may use both a left and right axis.  The color or
% style and axis for each measure is given in codes in parentheses in
% the title of the plot.  The first code abbreviates the color or line
% style:
%
%      R  red          -   solid
%      G  green        --  dashed
%      B  blue         :   dotted
%      C  cyan
%      M  magenta
%      Y  yellow
%      K  black
%      W  white
%
% and the second letter indicates the axis, L for left and R for
% right.
%
if indVarLen > 1
  figure (figSum);
  clf;
  axesSum = zeros (12, 1);  % space for axes handles
  %
  % Axes 1: pointing error
  %
  subplot (4, 2, 1);
  axesSum (1) = gca;
  if any (numAcc > 1)
    hline = errorbar (indVar, errPointS (:, 1)  / rpd, sqrt (errPointS (:, 2)) / rpd);
    set (hline, discrim, discrimValue {1});  % possibly override errorbar's default solid line style
    set (hline (1), 'linestyle', '-');       %   but leave the error bars themselves solid
  else
    plot (indVar, errPointS (:, 1) / rpd);
  end
  ylabel ('(deg)');
  title ('pointing error');
  %
  % Axes 2 and 3: beam power and directivity
  %
  subplot (4, 2, 2);
  axesSum (2) = gca;
  if any (numAcc > 1)
    hline = errorbar (indVar, beamPowerDBS (:, 1), sqrt (beamPowerDBS (:, 2)));
    set (hline, discrim, discrimValue {1});
    set (hline (1), 'linestyle', '-');
  else
    plot (indVar, beamPowerDBS (:, 1));
  end
  ylabel ('(dB re coherent)');
  title (strcat ('peak power dens (', discrimName {1}, ...
                 ' L), directivity (', discrimName {2}, ' R)'));
  axesSum (3) = axes ('position', get (gca, 'position'));
  if any (numAcc > 1)
    hline = errorbar (indVar, directivityDBS (:, 1), sqrt (directivityDBS (:, 2)));
    set (hline, discrim, discrimValue {2});
    set (hline (1), 'linestyle', '-');
  else
    plot (indVar, directivityDBS (:, 1), discrim, discrimValue {2});
  end
  set (gca, 'yAxisLocation', 'right', 'color', 'none');
  %
  % Axes 4 and 5: beam widths
  %
  subplot (4, 2, 3);
  axesSum (4) = gca;
  if any (numAcc > 1)
    hline = errorbar (indVar, hpbwMjrS (:, 1)  / rpd, sqrt (hpbwMjrS (:, 2)) / rpd);
    set (hline, discrim, discrimValue {1});
    set (hline (1), 'linestyle', '-');
  else
    plot (indVar, hpbwMjrS (:, 1) / rpd);
  end
  ylabel ('(deg)');
  title (strcat ('hpbw major (', discrimName {1}, ...
                 ' L) & minor (', discrimName {2}, ' R)'));
  axesSum (5) = axes ('position', get (gca, 'position'));
  if any (numAcc > 1)
    hline = errorbar (indVar, hpbwMnrS (:, 1)  / rpd, sqrt (hpbwMnrS (:, 2)) / rpd);
    set (hline, discrim, discrimValue {2});
    set (hline (1), 'linestyle', '-');
  else
    plot (indVar, hpbwMnrS (:, 1) / rpd, discrim, discrimValue {2});
  end
  set (gca, 'yAxisLocation', 'right', 'color', 'none');
  yLimL = get (axesSum (4), 'ylim');
  yLimR = get (axesSum (5), 'ylim');
  if yLimL (1) < yLimR (2)
    set (axesSum (4), 'xlimmode', 'manual', 'ylim', [yLimR(1) yLimL(2)]);
    set (axesSum (5), 'xlimmode', 'manual', 'ylim', [yLimR(1) yLimL(2)]);
```

```
    % Setting xlimmode to manual prevents rescaling of the x axis when
    % the y axis is changed.
  end
  %
  % Axes 6: beam roll
  %
  subplot (4, 2, 4);
  axesSum (6) = gca;
  if any (numAcc > 1)
    hline = errorbar (indVar, rollS (:, 1)  / rpd, sqrt (rollS (:, 2)) / rpd);
    set (hline, discrim, discrimValue {1});
    set (hline (1), 'linestyle', '-');
  else
    plot (indVar, rollS (:, 1) / rpd);
  end
  ylabel ('(deg)');
  title ('beam roll');
  %
  % Axes 7: beam depth
  %
  subplot (4, 2, 5);
  axesSum (7) = gca;
  if any (numAcc > 1)
    hline = errorbar (indVar, beamDepthDBS (:, 1), sqrt (beamDepthDBS (:, 2)));
    set (hline, discrim, discrimValue {1});
    set (hline (1), 'linestyle', '-');
  else
    plot (indVar, beamDepthDBS (:, 1));
  end
  ylabel ('(dB re peak)');
  title ('beam depth');
  %
  % Axes 8 and 9: power ratios
  %
  subplot (4, 2, 6);
  axesSum (8) = gca;
  if any (numAcc > 1)
    hline = errorbar (indVar, powerMainVisbDBS(:,1), sqrt (powerMainVisbDBS(:,2)));
    set (hline, discrim, discrimValue {1});
    set (hline (1), 'linestyle', '-');
  else
    plot (indVar, powerMainVisbDBS(:,1));
  end
  ylabel ('(dB)');
  title (strcat ('power m/v (', discrimName {1}, ...
                 ' L), m/s (', discrimName {2}, ...
                 ' R), v/s (', discrimName {3}, ' R)'));
  axesSum (9) = axes ('position', get (gca, 'position'));
  if any (numAcc > 1)
    hline = errorbar (indVar, powerMainSideDBS(:,1), ...
                      sqrt (powerMainSideDBS(:,2)));
    set (hline, discrim, discrimValue {2});
    set (hline (1), 'linestyle', '-');
    set (gca, 'nextplot', 'add');
    hline = errorbar (indVar, powerVisbSideDBS(:,1), ...
                      sqrt (powerVisbSideDBS(:,2)));
    set (hline, discrim, discrimValue {3});
    set (hline (1), 'linestyle', '-');
    set (gca, 'nextplot', 'replace');
  else
    plot (indVar, powerMainSideDBS(:,1), discrim, discrimValue {2});
    set (gca, 'nextplot', 'add');
    plot (indVar, powerVisbSideDBS(:,1), discrim, discrimValue {3});
    set (gca, 'nextplot', 'replace');
  end
  set (gca, 'yAxisLocation', 'right', 'color', 'none');
  %
  % Axes 10 and 11: sidelobe powers
  %
  subplot (4, 2, 7);
  axesSum (10) = gca;
  if any (numAcc > 1)
    hline = errorbar (indVar, slLgstPowrDBS (:, 1), sqrt (slLgstPowrDBS (:, 2)));
    set (hline, discrim, discrimValue {1});
    set (hline (1), 'linestyle', '-');
    set (gca, 'nextplot', 'add');
    hline = errorbar (indVar, slNrstPowrDBS (:, 1), sqrt (slNrstPowrDBS (:, 2)));
    set (hline, discrim, discrimValue {2});
    set (hline (1), 'linestyle', '-');
    set (gca, 'nextplot', 'replace');
  else
    plot (indVar * ones (1, 2), ...
      [slLgstPowrDBS(:,1) slNrstPowrDBS(:,1)]);
  end
  ylabel ('(dB re peak)');
  title (strcat ('sidelobe power: lgst (', discrimName {1}, ...
                            ' L), nrst (', discrimName {2}, ...
                            ' L), avg (',  discrimName {3}, ' R)'));
  axesSum (11) = axes ('position', get (gca, 'position'));
  if any (numAcc > 1)
    hline = errorbar (indVar, powerSideAvgDBS (:, 1), sqrt (powerSideAvgDBS (:, 2)));
    set (hline, discrim, discrimValue {3});
```

```
      set (hline (1), 'linestyle', '-');
    else
      plot (indVar, powerSideAvgDBS (:, 1), discrim, discrimValue {3});
    end
    set (gca, 'yAxisLocation', 'right', 'color', 'none');
    %
    % Axes 12: sidelobe distances
    %
    subplot (4, 2, 8);
    axesSum (12) = gca;
    if any (numAcc > 1)
      hline = errorbar (indVar, slLgstDistS (:, 1) / rpd, sqrt (slLgstDistS (:, 2)) / rpd);
      set (hline, discrim, discrimValue {1});
      set (hline (1), 'linestyle', '-');
      set (gca, 'nextplot', 'add');
      hline = errorbar (indVar, slNrstDistS (:, 1) / rpd, sqrt (slNrstDistS (:, 2)) / rpd);
      set (hline, discrim, discrimValue {2});
      set (hline (1), 'linestyle', '-');
      set (gca, 'nextplot', 'replace');
    else
      plot (indVar * ones (1, 2), ...
        [slLgstDistS(:,1) slNrstDistS(:,1)] / rpd);
    end
    ylabel ('(deg)');
    title (strcat ('sidelobe distance: lgst (', discrimName {1}, ...
                                    '), nrst (', discrimName {2}, ')'));
    %
    % Touch up
    %
    set (findobj (gcf, 'type', 'axes'), 'xlim', ...      % make x-axis limits uniform
      [min(indVar) max(indVar)] ...
      + 0.1 * (max (indVar) - min (indVar)) * [-1 1]);
    axesSumTitle = axes ('position', [0 0 1 1], ...      % create invisible axes
      'color', 'none', 'visible', 'off', ...             %   for titling
      'defaultTextFontSize', 10, ...
      'defaultTextHorizontalAlignment', 'center');
    text (0.5, 0.05, indVarName, ...                     % display name of independent
      'horizontalAlignment', 'center');                  %   variable
  end
  clear hline

  % Plot last realization
  %
  % Three projections of the hemisphere are available: Lambert,
  % stereographic, and orthographic.
  %
  % Lambert projection:
  %
  % The Lambert projection preserves the relative areas of portions of
  % the hemisphere.  That is, the ratio of areas of two regions on the
  % projection is the same as on the hemisphere.  The azimuth coordinate
  % of a point in the projection is the same as its azimuth coordinate
  % on the hemisphere, while its radius  r  from the center of the
  % projection is related to the polar angle.  This relationship may be
  % derived by setting the spherical surface area  sin polar d(polar)
  % d(azmth)  equal to a constant times the planar surface area  r dr
  % d(azmth)  and integrating.  The radius is then given by
  %
  %            1/2     polar
  %       r = 2    sin ----- .
  %                      2
  %
  % For computer graphics the Cartesian coordinates are more convenient;
  % they are
  %
  %       u = r cos azmth = R cx
  %
  %       v = r sin azmth = R cy
  %
  % where
  %
  %            -1/2     polar            -1/2
  %       R = 2      sec ----- = (1 + cz)
  %                        2
  %
  % For points outside visible space (that is, for cx^2 + cy^2 > 1), cz
  % is zero so that R = 1 and no transformation is applied.
  %
  % Stereographic projection:
  %
  % The stereographic projection preserves angles on the hemisphere.  To
  % derive the governing relationship for the projection, use the same
  % azimuthal angle for the projection as for the hemisphere, and let
  % the radius be a function of the polar angle: r = f (polar).  Equate
  % the aspect ratios of orthogonal derivatives, as
  %
  %           d(polar)              dr
  %       ------------------ = ----------
  %       sin polar d(azmth)    r d(azmth)
  %
```

```
%                            f'(polar) d(polar)
%                          = ------------------
%                            f (polar) d(azmth)
%
 (where f' is the first derivative), rearrange, and integrate to
% obtain
%
%                       polar   1 - cos polar
%      f (polar) = tan ----- = ------------- ,
%                         2       sin polar
%
% up to an arbitrary multiplicative constant.  The projected Cartesian
% coordinates of a point (cx, cy, cz) on the sphere are
%
%      u = r cos azmth = R cx
%
%      v = r sin azmth = R cy
%
% where
%
%          f (polar)         1            1
%      R = --------- = ------------- = ------ .
%          sin polar   1 + cos polar   1 + cz
%
% For points outside visible space (that is, for cx^2 + cy^2 > 1), cz
% is zero so that R = 1 and no transformation is applied.
%
% The center of projection is opposite boresight (cx = cy = 0, cz =
% -1), and with the above choice of multiplicative constant, the plane
% of projection is the cz = 0 plane.
%
% Orthographic projection:
%
% The orthographic projection gives a 3D view of the hemisphere.
%
if 1                        % INPUT 1 to plot, 0 to skip
  proj = 1;                 % INPUT 1 for 2D equal-area Lambert
                            %       2 for 2D stereographic
                            %       3 for orthographic (3D hemisphere)
  coordSys = 1;             % INPUT 1 for grid of spherical coordinates
                            %       2 for grid of traditional coordinates
  faceColor = 'flat';       % INPUT 'flat' or 'interp' shading
  pointZoom =  0;           % INPUT magnification factor or 0 for centered full view
  % The "show" inputs below are coded only for 2D views.
  showBeamRegion    = 0;  % INPUT 1 to show main beam region
  showWidthRegion   = 0;  %                 region above width contour
  showWidthContAct = 0;  %                 actual width contour
  showWidthContFit = 0;  %                 fitted width contour (ellipse)
  showPointGrid     = 0;  %                 main beam peak on sampled grid
  showPoint         = 1;  %                 interpolated main beam peak (pointing vector)
  showPointUnc      = 0;  %                 axes of uncertainty ellipse of pointing vector
  showSidelobeGrid = 0;  %                 sidelobe peaks on sampled grid
  showSidelobeNrst = 1;  %                 nearest sidelobe
  showSidelobeLgst = 1;  %                 largest sidelobe
  %
  % Prepare for plotting
  %
  if strcmp (faceColor, 'interp')
    dirCosXMtxSurf = dirCosXMtx;                        % use true grid
    dirCosYMtxSurf = dirCosYMtx;
    dirCosZMtxSurf = dirCosZMtx;
    visBoolSurf    = visBool    ;
  elseif strcmp (faceColor, 'flat')
    dirCosXMtxSurf = ones (ay, 1) * dirCosShiftX;  % use shifted grid to center
    dirCosYMtxSurf = dirCosShiftY * ones (1, ax);  %   patches on data points
    radSqr = dirCosXMtxSurf.^2 + dirCosYMtxSurf.^2;
    visBoolSurf = (radSqr < 1);
    dirCosZMtxSurf = zeros (ay, ax);
    dirCosZMtxSurf (visBoolSurf) = sqrt (1 - radSqr (visBoolSurf));
    clear radSqr;
  else
    error ('Illegal value of faceColor.');
  end
  if strcmp (faceColor, 'interp')             % identify visible points plus those immediately
    gBlnkIndx = zeros (ay, ax);               %   or diagonally adjacent (Boolean for now)
    gBlnkIndx (1:ay-1, 1:ax-1) = visBool (2:ay, 2:ax);
    gBlnkIndx (1:ay-1, 2:ax  ) = gBlnkIndx (1:ay-1, 2:ax  ) | visBool (2:ay  , 1:ax-1);
    gBlnkIndx (2:ay  , 1:ax-1) = gBlnkIndx (2:ay  , 1:ax-1) | visBool (1:ay-1, 2:ax  );
    gBlnkIndx (2:ay  , 2:ax  ) = gBlnkIndx (2:ay  , 2:ax  ) | visBool (1:ay-1, 1:ax-1);
  else                                        % identify visible points plus those immediately
    gBlnkIndx = visBoolSurf;                  %   but not diagonally adjacent (Boolean for now)
    gBlnkIndx (1:ay-1, 1:ax  ) = gBlnkIndx (1:ay-1, 1:ax  ) | visBoolSurf (2:ay  , 1:ax);
    gBlnkIndx (1:ay-1, 1:ax-1) = gBlnkIndx (1:ay-1, 1:ax-1) | visBoolSurf (2:ay  , 2:ax);
    gBlnkIndx (1:ay  , 1:ax-1) = gBlnkIndx (1:ay  , 1:ax-1) | visBoolSurf (1:ay  , 2:ax);
  end
  gBlnkIndx = ~gBlnkIndx;                     % 0 in visible region plus appropriate border
  gZeroIndx = (gSqr == 0);                    % 1 where zero, 0 elsewhere (Boolean for now)
  gOKIndx = find (~(gBlnkIndx | gZeroIndx));  % 1 for nontrivial values
  gZeroIndx = find (gZeroIndx);               % convert to indices
  gBlnkIndx = find (gBlnkIndx);
  if 1                                        % INPUT 0 for linear, 1 for decibel plot
    % Decibel plot
```

```
      gPlot = zeros (ay, ax);
      clim = [-50 0];                              % INPUT dB range
      gPlot (gOKIndx) = max (clim (1), 10 * log10 (gSqr (gOKIndx)));
      gPlot (gZeroIndx) = clim (1) * ones (size (gZeroIndx));  % condition log of 0
    else
      % Linear plot
      gPlot = gSqr;
      clim = [0 1];
    end
    if invertBkgd                                   % set appropriate value for invisible points
      gPlot (gBlnkIndx) = clim (2) * ones (size (gBlnkIndx));
    else
      gPlot (gBlnkIndx) = clim (1) * ones (size (gBlnkIndx));
    end
    clear visBoolSurf gBlnkIndx gZeroIndx gOKIndx;
    figure (figPat);
    fore = get (figPat, 'defaultLineColor');     % expect black or white
    if invertBkgd  % choose fore- and background colors compatible with color map
      fore = 1 - fore;                          % swap
    end
    back = 1 - fore;
    switch coordSys
      case 1
        grid1Polar = (0 : 10 :  90) * rpd;      % lines of constant polar angle
        grid1Azmth = (0 :  5 : 360) * rpd - azmthOffst;
        grid2Polar = (0 :  5 :  90) * rpd;      % lines of constant azimuth
        grid2Azmth = (0 : 30 : 360) * rpd - azmthOffst;
        [grid1Polar, grid1Azmth] = meshgrid (grid1Polar, grid1Azmth);
        [grid2Azmth, grid2Polar] = meshgrid (grid2Azmth, grid2Polar);
      case 2
        grid1Az = (-90 : 10 : 90) * rpd;        % lines of constant azimuth
        grid1El = (-90 :  5 : 90) * rpd;
        grid2Az = (-90 : 10 : 90) * rpd;        % lines of constant elevation
        grid2El = (-90 : 10 : 90) * rpd;
        [grid1Az, grid1El] = meshgrid (grid1Az, grid1El);
        [grid2El, grid2Az] = meshgrid (grid2El, grid2Az);
        grid1Polar = acos  (cos (grid1El) .* cos (grid1Az));
        grid1Azmth = atan2 (tan (grid1El),  -sin (grid1Az)) - azmthOffst;
        grid2Polar = acos  (cos (grid2El) .* cos (grid2Az));
        grid2Azmth = atan2 (tan (grid2El),  -sin (grid2Az)) - azmthOffst;
      otherwise
        error ('Invalid value of coordSys.');
    end
    %
    % Plot array factor
    %
    if proj == 1
      %
      % Lambert projection
      %
      radFactSurf = 1 ./ sqrt (1 + dirCosZMtxSurf);  % radius factor for surface plot
      hSurf = surf (radFactSurf .* dirCosXMtxSurf, ...
                    radFactSurf .* dirCosYMtxSurf, gPlot);
      grid1RadFact = sqrt (2) * sin (grid1Polar / 2);
      line (grid1RadFact .* cos (grid1Azmth), ...
            grid1RadFact .* sin (grid1Azmth), ...
            clim (2) * ones (size (grid1RadFact)), 'color', fore);
      grid2RadFact = sqrt (2) * sin (grid2Polar / 2);
      line (grid2RadFact .* cos (grid2Azmth), ...
            grid2RadFact .* sin (grid2Azmth), ...
            clim (2) * ones (size (grid2RadFact)), 'color', fore);
      set (gca, 'drawmode', 'fast');                  % no hidden objects to worry about
      xylim = [-1 1];
      zlim = clim;
      viewAz = 0;                                     % Matlab azimuth and
      viewEl = 90 * rpd;                              %   elevation (but in radians)
    elseif proj == 2
      %
      % Stereographic projection
      %
      radFactSurf = 1 ./ (1 + dirCosZMtxSurf);       % radius factor for surface plot
      hSurf = surf (radFactSurf .* dirCosXMtxSurf, ...
                    radFactSurf .* dirCosYMtxSurf, gPlot);
      grid1RadFact = tan (grid1Polar / 2);
      line (grid1RadFact .* cos (grid1Azmth), ...
            grid1RadFact .* sin (grid1Azmth), ...
            clim (2) * ones (size (grid1RadFact)), 'color', fore);
      grid2RadFact = tan (grid2Polar / 2);
      line (grid2RadFact .* cos (grid2Azmth), ...
            grid2RadFact .* sin (grid2Azmth), ...
            clim (2) * ones (size (grid2RadFact)), 'color', fore);
      set (gca, 'drawmode', 'fast');                  % no hidden objects to worry about
      xylim = [-1 1];
      zlim = clim;
      viewAz = 0;                                     % Matlab azimuth and
      viewEl = 90 * rpd;                              %   elevation (but in radians)
    elseif proj == 3
      %
      % Orthographic projection (3D hemisphere)
      %
      float = 1.02;  % radius of annotations relative to unit hemisphere
      hSurf = surf (dirCosXMtxSurf, dirCosYMtxSurf, dirCosZMtxSurf, gPlot);
```

```
      grid1RadFact = sin (grid1Polar);
      line (grid1RadFact .* cos (grid1Azmth) * float, ...
            grid1RadFact .* sin (grid1Azmth) * float, ...
            cos (grid1Polar) * float, 'color', fore);
      grid2RadFact = sin (grid2Polar);
      line (grid2RadFact .* cos (grid2Azmth) * float, ...
            grid2RadFact .* sin (grid2Azmth) * float, ...
            cos (grid2Polar) * float, 'color', fore);
      set (gca, 'drawmode', 'normal');              % remove hidden objects
      xylim = [-1 1] * float;
      zlim  = [-1 1] * float;
      switch 1  % INPUT 1 to look down main beam; 2, down boresight; 3, custom
        case 1
          % Look down main beam
          viewAz = pi/2 + steerAzmth + azmthOffst;  % Matlab azimuth and
          viewEl = pi/2 - steerPolar;               %   elevation (but in radians)
        case 2
          % Look down boresight
          viewAz = 0;                               % Matlab azimuth and
          viewEl = 90 * rpd;                        %   elevation (but in radians)
          set (gca, 'drawmode', 'fast');            % no hidden surfaces
        otherwise
          % Look somewhere
          viewAz = 60 * rpd;  % INPUT custom view azimuth
          viewEl = 30 * rpd;  %   and elevation (Matlab coordinates)
      end
    end  % plotting
    %
    % Annotate plot (2D only)
    %
    if (proj == 1) | (proj == 2)
      switch proj  % set appropriate radius factor
        case 1
          radFact = 1 ./ sqrt (1 + dirCosZMtx);
        case 2
          radFact = 1 ./ (1 + dirCosZMtx);
      end
      if showBeamRegion & beamExist
        line (radFact (beamIndx) .* dirCosXMtx (beamIndx), ...
              radFact (beamIndx) .* dirCosYMtx (beamIndx), ...
              clim (2) * ones (size (beamIndx)), ...
              'linestyle', 'none', 'marker', '+', 'color', fore);
      end
      if showWidthRegion & capClosed
        line (radFact (capIndx) .* dirCosXMtx (capIndx), ...
              radFact (capIndx) .* dirCosYMtx (capIndx), ...
              clim (2) * ones (size (capIndx)), ...
              'linestyle', 'none', 'marker', 'x', 'color', fore);
      end
      if showWidthContAct & capClosed
        switch proj
          case 1
            radFactCapAct = 1 ./ sqrt (1 + capContZ);
          case 2
            radFactCapAct = 1 ./ (1 + capContZ);
        end
        line (radFactCapAct .* capContX, radFactCapAct .* capContY, ...
          clim (2) * ones (size (capContX)), 'linestyle', ':', 'color', back);
        clear radFactCapAct
      end
      if showWidthContFit & capClosed
        switch proj
          case 1
            radFactCapFit = 1 ./ sqrt (1 + cFitZ);
          case 2
            radFactCapFit = 1 ./ (1 + cFitZ);
        end
        line (radFactCapFit .* cFitX, radFactCapFit .* cFitY, ...
          clim (2) * ones (size (cFitX)), 'linestyle', '-', 'color', back);
        clear radFactCapFit
      end
      if showPointGrid
        line (radFact (gSqrMaxRow, gSqrMaxCol) .* dirCosXMtx (gSqrMaxRow, gSqrMaxCol), ...
              radFact (gSqrMaxRow, gSqrMaxCol) .* dirCosYMtx (gSqrMaxRow, gSqrMaxCol), ...
              clim (2), 'linestyle', 'none', 'marker', '*', 'color', back);
      end
      if showPointUnc
        switch proj
          case 1
            pointRadFact = 1 / sqrt (1 + pz);
            Jacob = [1+pz+px^2/pz       px*py/pz                          % Jacobian of
                           py*px/pz  1+pz+py^2/pz] / (2*(1+pz)^(3/2));    %   projection
          case 2
            pointRadFact = 1 / (1 + pz);
            Jacob = [1+pz+px^2/pz       px*py/pz                          % Jacobian of
                           py*px/pz  1+pz+py^2/pz] / (1+pz)^2;            %   projection
        end
        pVarProj = Jacob * pVar * Jacob';  % covariance matrix in this projection
        [pEigVec, pEigVal] ...             % diagonalize: pVar
          = eig (pVarProj);                %   = pEigVec * pEigVal * pEigVec'
        pPrAx = 2 * pEigVec ...            % scale principal axes (columns of
          * sqrt (pEigVal);                %   pEigVec) to 2 standard deviations
```

```
    line (pointRadFact * px + [-1 1]' * pPrAx (1, :), ...
          pointRadFact * py + [-1 1]' * pPrAx (2, :), ...
          clim (2) * [1 1], ...
          'linestyle', '-', 'color', back);
    clear pointRadFact Jacob pVarProj pEigVec pEigVal pPrAx
  elseif showPoint
    switch proj
      case 1
        pointRadFact = 1 / sqrt (1 + pz);
      case 2
        pointRadFact = 1 / (1 + pz);
    end
    line (pointRadFact * px, pointRadFact * py, clim (2), ...
      'linestyle', 'none', 'marker', '.', 'color', back)
    clear pointRadFact
  end
  if showSidelobeGrid
    line (radFact (slIndx) .* dirCosXMtx (slIndx), ...
          radFact (slIndx) .* dirCosYMtx (slIndx), ...
          clim (2) * ones (size (slIndx)), ...
          'linestyle', 'none', 'marker', 's', 'color', back);
  end
  if showSidelobeNrst
    switch proj
      case 1
        radFactSlNrst = 1 / sqrt (1 + slNrstVec (3));
      case 2
        radFactSlNrst = 1 / (1 + slNrstVec (3));
    end
    line (radFactSlNrst .* slNrstVec (1), ...
          radFactSlNrst .* slNrstVec (2), ...
          clim (2), 'linestyle', 'none', 'marker', '+', 'color', back);
    clear radFactSlNrst
  end
  if showSidelobeLgst
    switch proj
      case 1
        radFactSlLgst = 1 / sqrt (1 + slLgstVec (3));
      case 2
        radFactSlLgst = 1 / (1 + slLgstVec (3));
    end
    line (radFactSlLgst .* slLgstVec (1), ...
          radFactSlLgst .* slLgstVec (2), ...
          clim (2), 'linestyle', 'none', 'marker', 'x', 'color', back);
    clear radFactSlLgst
  end
end  % annotations
%
% Arrange graphics properties
%
axesPat = gca;
rotate (get (axesPat, 'children'), ...  % rotate plotted objects to
  [0 0 1], azmthOffst / rpd);           %   compensate for azmthOffst
set (hSurf, 'edgecolor', 'none');
set (hSurf, 'facecolor', faceColor);
colormap (cmap);
set (axesPat, 'clim', clim);
if cbarVert                         % position the colorbar
  axesCbar = colorbar ('vert');
  set (axesCbar, ...
    'units', 'normalized', ...
    'position', [(1+0.2*cbarSize)/(1+cbarSize) 0.05 0.4*cbarSize/(1+cbarSize) 0.9]);
  set (axesPat, ...
    'units', 'normalized', ...
    'position', [0 0 1/(1+cbarSize) 1]);
else
  axesCbar = colorbar ('horiz');
  set (axesCbar, ...
    'units', 'normalized', ...
    'position', [0.05 0.3*cbarSize/(1+cbarSize) 0.9 0.5*cbarSize/(1+cbarSize)]);
  set (axesPat, ...
    'units', 'normalized', ...
    'position', [0 cbarSize/(1+cbarSize) 1 1/(1+cbarSize)]);
end
set (figPat, 'children', ...  % put color bar in front of pattern but let
  [axesCbar axesPat]');       %   axesPat remain the current axis
set (axesPat, ...
  'xlim', xylim, ...
  'ylim', xylim, ...
  'zlim', zlim , ...
  'dataAspectRatio', diff ([xylim' xylim' zlim']), ...
  'visible', 'off', ...
  'view', [viewAz/rpd  viewEl/rpd]);
if proj == 3
  cameraDist = norm ( ...                        % distance from camera
      get (axesPat, 'cameraPosition') ...        %   to the surface
    - get (axesPat, 'cameraTarget'), 2);
  set (axesPat, 'cameraViewAngle', ...           % set view angle to
    2 * atan (float * diff (zlim) ...            %   exactly span the
            / (cameraDist * diff (xylim))) / rpd);  %   unit sphere
end
if (pointZoom ~= 0) & peakVisb
```

```
        pxRot = cos (azmthOffst) * px - sin (azmthOffst) * py;  % compensate for
        pyRot = sin (azmthOffst) * px + cos (azmthOffst) * py;  %   azmthOffst
        if proj == 1
          pointRadFact = 1 / sqrt (1 + pz);
          set (axesPat, 'xlim', pointRadFact * pxRot + [-1 1] / pointZoom, ...
                        'ylim', pointRadFact * pyRot + [-1 1] / pointZoom);
        elseif proj == 2
          pointRadFact = 1 / (1 + pz);
          set (axesPat, 'xlim', pointRadFact * pxRot + [-1 1] / pointZoom, ...
                        'ylim', pointRadFact * pyRot + [-1 1] / pointZoom);
        elseif proj == 3
          set (axesPat, 'cameraViewAngle', ...
            2 * atan (float / (cameraDist * pointZoom)) / rpd);
        end
        clear pxRot pyRot pointRadFact
      end  % zoom
      clear dirCosXMtxSurf dirCosYMtxSurf dirCosZMtxSurf;
      clear fore back gridPolar gridAzmth gridAzmthSmth;
      clear radFactSurf gridFact viewAz viewEl float radFact;
    end

    drawnow;

  end  % loop over independent variable
  clear indx

  % Print warnings, if any
  %
  if any (numAcc < numRlz)
    disp ('Warning: at least one realization could not be fully');
    disp ('  analyzed; inspect the matfiles in the current directory.');
  end

  clear cbarSize
  clear lx ly mx my nx ny;
  clear numLMX numLMY numLMNX numLMNY;
  clear tIndx rlzNum;

  % end of program

  % Suggestions for improvements
  %
  % Implement non-uniform excitation magnitudes:
  %
  %   As needed, non-uniform illumination weights may be coded
  %   straightforwardly.
  %
  % Implement element factor and improve integration of radiated powers:
  %
  %   Currently the element factor is implicitly coded as one everywhere,
  %   so that the elements radiate uniformly into the hemisphere.  A more
  %   realistic element factor is cos polar [1], being one at broadside
  %   and zero at grazing.  Note, however, that in integrating the data
  %   over solid angles to calculate radiated powers, we divide the data
  %   by cos polar.  Performing this multiplication and division in
  %   succession could yield invalid data near grazing, where cos polar is
  %   small. Obviously, this difficulty could be avoided by maintaining
  %   the unscaled data for use in the integration while using the scaled
  %   data for all other processing.  The calculations of the pointing
  %   vector from the excitation phases and of the peak power should also
  %   account for the element factor.
  %
  %   More generally, one might wish to apply an arbitrary element factor.
  %   If it is small near grazing, the difficulty described above
  %   persists.  One solution is to specify the element factor relative to
  %   cos polar; the user ensures that all values of that ratio are
  %   reasonable.  The data used in the integration are scaled by the
  %   ratio, while the data used elsewhere are scaled by both the ratio
  %   and cos polar.  One could also allow element factors defined over
  %   the entire sphere, including radiation into z < 0, perhaps by
  %   storing the back radiation pattern in a second g matrix and
  %   modifying the analysis routines.
  %
  %   Broadening our perspective, we note that these problems are
  %   ultimately due to the integration algorithm, in that it blindly
  %   applies a Jacobian that diverges at grazing.  One consequence is
  %   that data cells whose centers are just inside the border of visible
  %   space contribute their entire value scaled by a large Jacobian,
  %   whereas those whose centers are just outside contribute nothing.
  %   More correctly, both should contribute about half of their value
  %   scaled according to some average location of the contributing
  %   region.  Cells almost entirely outside of visible space should
  %   contribute very little.  A better algorithm would reproduce this
  %   behavior.
  %
  % Allow pattern cuts:
  %
  %   Often one is interested in a cut of the pattern along a path on the
  %   unit hemisphere, such as cuts through the main beam along azimuth
  %   and elevation curves or along the great circles of maximum and
  %   minimum beam width.  If only a graphical presentation is desired,
  %   the cut could be interpolated from the pattern.  If an analysis is
```

```
%  also desired, specialized routines would probably be required, as
%  the current routines expect data over two directional coordinates.
%
% Consider alternate calculation of beam widths and roll:
%
%  The beam widths and roll are derived from an ellipse fitted to a
%  level contour of the main beam.  Currently the fit is performed in
%  the boresight-centered stereographic projection regardless of the
%  pointing vector.  This projection preserves the orientation and
%  orthogonality of the major and minor axes of the ellipse.  However,
%  because scale in the stereographic projection increases away from
%  boresight, off-boresight contours are expanded toward the edge of
%  the projection.  Although scale is uniform in all directions for an
%  infinitesimal region, radial scale is exaggerated relative to
%  azimuthal scale for a finite region.  For a broad beam off
%  boresight, the distortion of beam width may be significant.
%
%  An improvement might be achieved with two changes.  First, center
%  the projection on the pointing vector, assuming that the contour
%  will be found to lie centered on the pointing vector as well.
%  Second, instead of the stereographic projection, use the azimuthal
%  equidistant projection, for which distances measured on a line
%  passing through the center of the projection are true.  Regardless
%  of beam width, the lengths of the major and minor axes of the
%  projected ellipse will equal those of the ellipse on the hemisphere,
%  provided that the center of the ellipse is also the center of the
%  projection.
%
%  In centering the projection on the pointing vector, one is free to
%  choose the orientation of the projection relative to the spherical
%  coordinate system, and a judicious choice of this angle facilitates
%  calculation of the beam's roll angle.  Construct at boresight on the
%  hemisphere two tangent axes u and v; let positive u be directed
%  toward spherical azimuth zero and positive v toward pi/2.  Next,
%  construct the arc connecting boresight with the pointing vector.
%  Translate the u-v origin and system along this arc without rotation
%  in the plane locally tangent to the hemisphere, so that u, v, and
%  the arc maintain the same local orientation.  If the ellipse is
%  centered on the pointing vector, the roll angle will be the angle
%  from the positive u axis to the major axis.
%
% Allow linear arrays:
%
%  The current analysis routines cannot handle linear arrays because of
%  several incompatibilities.  For example, the main beam of
%  one-dimensional arrays with isotropic element patterns is a cone
%  about the axis of the array, so the direction of radiation is
%  specified by a single number -- the angle between the axis and the
%  cone.  However, the current analysis routines seek a two-parameter
%  specification of the main beam direction and will generally fail.
%  Likewise, the beam width is described by one number, but the current
%  routines seek two parameters.  Furthermore, analyses that depend on
%  these values (such as determining the proximity of sidelobes) will
%  also fail.  If linear arrays are if interest, the analysis routines
%  must be expanded.  One might also add graphics routines tailored to
%  one-dimensional arrays.
%
%  Note that a non-isotropic element pattern will generally produce a
%  variation in the direction orthogonal to the main beam contour,
%  allowing the analysis routines to proceed. Results thereby obtained
%  should be interpreted accordingly.  (Using a non-isotropic element
%  pattern will not benefit the routine that locates the pointing
%  vector from the excitation phases.  One might ignore its results
%  when the final pointing vector is calculated, using only the
%  pointing vector obtained from the Fourier transform.)
%
%  [1] R. Tang and R. W. Burns, "Phased Arrays," in _Antenna
%  Engineering Handbook_, 3rd ed., R. C. Johnson, Ed.  New York, NY:
%  McGraw-Hill, 1993, Ch. 20, Sec. 3.
```

REFERENCES

1. R. J. Mailloux, "Array grating lobes due to periodic phase, amplitude, and time delay quantization," *IEEE Trans. Antennas and Propagation*, vol. 32, pp. 1364–1368, Dec. 1984.
2. A. P. Goutzoulis and D. K. Davies, "Switched fiber optic delay architectures," in *Photonic Aspects of Modern Radar*, H. Zmuda and E. N. Toughlian, Eds. Boston, MA: Artech House, 1994, pp. 351–380.
3. A. K. Agrawal and E. L. Holzman, "Beamformer architectures for active phased-array radar antennas," *IEEE Trans. Antennas and Propagation*, vol. 47, pp. 432–442, Mar. 1999.
4. R. J. Mailloux, *Phased Array Antenna Handbook*, ch. 7. Norwood, MA: Artech House, 1994.
5. R. C. Hansen, *Phased Array Antennas*, Sec. 12.4. New York: John Wiley & Sons, 1998.
6. B. D. Steinberg, *Principles of Aperture and Array System Design*, ch. 13. New York: John Wiley & Sons, 1976.
7. B. P. Chrisman, "Planar array antenna design analysis," in *Proc. Tactical Communications Conf.*, 1990, vol. 1, pp. 705–731.
8. P. J. Wright, "Simulation of phased array antennas," in *Tenth International Conf. on Antennas and Propagation*, 1997, vol. 1, pp. 498–501.
9. A. K. Agrawal, E. L. Williamson, and J. G. Ferrante, "Design criteria for wideband active phased array antennas," in *IEEE Antennas and Propagation Society International Symposium 1997 Digest*, vol. 2, pp. 714–717.
10. J. J. Lee et al., "Photonic wideband array antennas," *IEEE Trans. Antennas and Propagation*, vol. 43, pp. 966-982, Sept. 1995.
11. The MathWorks, Inc., 3 Apple Hill Dr., Natick, MA 01760-2098.
12. J. P. Snyder, *Map Projections: A Working Manual*, US Geological Survey Professional Paper 1395. Washington, DC: United States Government Printing Office, 1987.
13. D. H. von Seggern, *CRC Standard Curves and Surfaces*. Boca Raton, FL: CRC, 1993.
14. R. S. Elliott, "Beamwidth and directivity of large scanning arrays," *Microwave J.*, vol. 7, pp. 74–82, Jan. 1964.
15. R. S. Elliott, "The Theory of Antenna Arrays," in *Microwave Scanning Antennas*, vol. 2, R. C. Hansen, Ed. New York: Academic, 1966, pp. 1–69.